Enabling Sender-initiated Distributed Applications and Checkpointing in Content Centric Networks

Nitinder Mohan

Computer Science and Engineering

Indraprastha Institute of Information Technology

Submitted in partial fulfillment of the requirements
for the Degree of ***Master of Technology***
with specialization in *Mobile Computing*

IIIT Delhi July 2015

Thesis Committee

1. **Dr. Pushpendra Singh** — Thesis Advisor

2. **Dr. Vinay Ribeiro** — External Examiner

3. **Dr. Vikram Goyal** — Internal Examiner

Date of Defense: 25 June, 2015

Signature of Post-Graduate Committee (PGC) Chair:

Keywords: Content Centric Networks; Sender-initiated communication; Checkpoint; CCN Application Checkpoint.

Dedicated to my elder sister, **Gagandeep**, who lost her life at a tender age of 15.
You will always be missed.

Abstract

Content Centric Network is a proposed future networking paradigm where data is the central entity for communication and the correspondence model follows two-step approach for data delivery. With increasing research in this domain, several new applications have been developed for CCN. However, the receiver-driven communication paradigm is unfavorable for several existing TCP/IP based distributed systems and would require extensive re-designing to support the proposed CCN architecture.

Checkpoint-restart is a widely used fault-recovery mechanism which saves the state of distributed system on a disk prior to a failure such that the system can restart from it. However, no prior work has been done till date to support checkpointing of distributed applications in CCN which is an absolute necessity if CCN is to be deployed on real-world infrastructure.

In this report, we present CCN Application Checkpoint (CCNAC) Tool, a plugin for checkpointing tool DMTCP, which enables checkpointing applications in CCN. We design and implement checkpointing algorithms to ensure consistent checkpoint-restart. We also propose a novel, efficient, application-layer based algorithm for sender-initiated communications based on proposed "pro-active naming" scheme in CCN. We design and evaluate test applications for our system on CCN testbed. Our proposed system is able to provide sender-initiated distributed computation with a nominal overhead of 28% and is able to checkpoint-restart both existing as well as proposed CCN distributed applications without much of communication overhead.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Nitinder Mohan
July 2015

Certificate

This is to certify that the thesis titled **"Enabling Sender-initiated Distributed Applications and Checkpointing in Content Centric Networks"** submitted by Nitinder Mohan for the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science & Engineering is a record of the bonafide work carried out by him under my guidance and supervision in the Mobile and Ubiquitous Computing group at Indraprastha Institute of Information Technology, Delhi. This work has not been submitted anywhere else for the reward of any other degree.

Dr. Pushpendra Singh
Indraprastha Institute of Information Technology (IIIT)
New Delhi

Acknowledgements

I would like to express my special appreciation and gratitude to my advisor Associate Professor Dr. Pushpendra Singh who has been a outstanding mentor for me. From him I learnt a persistent, open, and exploratory aptitude toward both research and life. His advice on both research as well as on my career has been priceless. I could not have imagined having a better advisor and mentor for my Master's research.

Besides my advisor, I would like to thank the esteemed committee members, Dr. Vinay Ribeiro and Dr. Vikram Goyal for agreeing to evaluate my work and thesis.

Special thanks are due to to my parents and my younger brother for their support and encouragement throughout my study. Last but not least I would like to thank all my friends for being with me at each step when I need their support. This thesis would never be successful without their love and support.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Peer-to-Peer distributed applications have become increasingly popular to solve high computation tasks over a network. A typical peer-to-peer application partitions tasks or workloads amongst equipotent participants/nodes. These applications are particularly useful when performing a task without a central server to coordinate between nodes. Such applications find there uses in various areas such as content delivery, multimedia, high computation cloud-based or cluster-based tasks etc.

However, such p2p applications are vulnerable to several faults at run time. A fault/ error in a system is deviation from the expected behavior of the system: a malfunction. Faults may be due to a variety of factors, including data corruption, hanging process, misleading return values, operator (user) error, and network problems [25]. Generally, a process needs to restart from the start after a failure to resume its consistent state. However, restart from start would be unfavorable for long running, high priority distributed process. To resume the process execution from the last known consistent computation, several failure recovery techniques are used such as checkpointing to backup disks, error correcting codes, packet retransmission, write-ahead logging etc. [2]. The most widely used failure recovery technique is checkpoint-restart.

Checkpoint-restart is an important mechanism to save the state of one or more running processes to a disk and later restore it after a failure. In addition to the traditional use case of fault tolerance in long-running jobs, other applications of checkpoint-restart include process migration, debugging, and save/restore of workspace. A long history of solutions have been proposed over the years to support checkpointing for distributed applications in TCP/IP networks [29, 31].

At a high-level, checkpointing a process can be viewed as writing all of process memory, including shared libraries, text and data, to a checkpoint image. Accordingly, restarting involves recreating the process memory by reading the checkpoint image from the disk. This

works for simple programs, but for complex programs, one also needs to save and restore information about threads, open files, etc. In more sophisticated applications, it involves saving the network state (in-flight data, etc.), and information about the external environment such as the terminal, the standard input/output/error, and so on [7].

The website checkpointing.org also lists several checkpoint-restart systems. There are three primary approaches to checkpointing: virtual machine snapshotting, application-level checkpointing, and transparent checkpointing. Several software packages have been been developed which can provide checkpoint-restart capability to distributed systems and desktop applications using these approaches. These packages are both kernel-level or user-level in nature [19, 32]. One such example of an open source, research-based, user-level checkpointing software is Distributed Multi-Threaded Checkpointing (DMTCP) [6].

Content Centric Networking (CCN) [21] is a proposed future Internet architecture that uses data names instead of host addresses for data delivery. The new architecture incorporates principles that have made the IP protocol suite widely adopted and globally scaled (e.g., the hourglass design and end-to-end principle), but changes the fundamental layer of the architecture to one better suited to modern networks and emerging communication patterns centrally dependent on data.

However, CCN follows a two-step, receiver-driven communication approach. This is an optimized approach for content-based applications such as sharing of content based resources, media streaming like VLC [1] etc. but fails in computation sharing sharing distributed applications. Most of these applications use a sender-initiated communication approach on TCP/IP networks and would not work well with receiver-driven architecture provided by CCN. Moreover, as we aim to deploy CCN over existing IP networks, porting of existing distributed applications over CCN would require developers to re-design their applications to suit the constraints imposed by CCN.

Also the existing checkpointing algorithms and tools fail to work due to the two-step communication model imposed by CCN [1]. However, as distributed applications on CCN are becoming more complex with time [18, 30, 35], a need for checkpointing these applications is necessary.

Keeping the above-mentioned motivations in mind, we present following contributions in this paper: (1) Provide a novel and efficient algorithms to sender-initiated communications in CCN. (2) Propose a concept of *pro-active names* which uses naming hierarchy of CCN packets to perform control actions at at a node. (3) Propose consistent checkpoint-restart algorithms for distributed applications on CCN Application Checkpoint (CCNAC) tool.

[1] existing checkpoint algorithms cater to single step, sender-driven TCP/IP model and fail to checkpoint two-step Interest-Data model of CCN. This was verified by us by checkpointing CCN-based application on DMTCP.

(4) Develop CCNAC, a plugin for DMTCP software which is capable of checkpointing applications on CCN. (5) Evaluated the performance of designed algorithms on both existing and proposed CCN distributed applications.

1.1 Research Aim

The research aim of the work presented in this report is the following:

1. Propose a novel and efficient approach to enable sender-initiated communication in CCN and implement it.

2. Understand various constraints involved while checkpointing applications in CCN and design CCN-optimized algorithms for checkpointing, if necessary.

1.2 Research Applications

The research work presented in this report has several applications. Few of them have been discussed.

1.2.1 Sender-driven Applications

Several applications involve sender-driven communications.

1. Distributed cloud-based applications.

2. Distributed cluster-based computation applications.

3. Publish-Subscribe networks for dissemination of data.

1.2.2 Checkpointing Applications

Several applications in CCN can use the checkpointing capability.

1. Distributed multiplayer games as developed by Qu et al. [30] and Wang et al. [35].

2. Online movie or video streaming applications

3. Remote Desktop using CCN

1.3 Organization

The remainder of the report has been organized as follows:

The related work to our research has been presented in Chapter 2. The chapter also contains various research contributions made by us through our research.

Chapter 3 presents the background related to Content Centric Networks, Checkpointing and its tools necessary to understand our contribution. This chapter explains the basics of routing in CCN and properties for consistent checkpoints which provides the motivation for our research.

Chapter 4 explains our proposal for sender-initiated approach of communication in detail. This is followed by design of hierarchical *pro-active* naming important to our approach.

Chapter 5 describes the various reasons for incompatibility of existing checkpointing algorithms in CCN. It further proposes CCNAC which incorporated new algorithms and rules to ensure consistent checkpoints in the system on top of existing TCP checkpointing using DMTCP. We also describe the format of the auxiliary data structures needed to checkpoint. Furthermore, the naming structure of checkpoint request and flush interests sent at time of checkpointing has also been explained,

Chapter 6 shows the implementation and the evaluation of the proposed technique to existing works. We also provide some in-depth analysis of the results and their possible implications in this chapter.

Finally, chapter 7 discusses about the limitations of the research in certain scenarios and paves a path for possible future work in our proposal.

Chapter 2

Related Work and Research Contribution

2.1 Related Work

Several similar works have been presented on the proposed research topic in this thesis. The work present in this report has several applications belonging to domains of publisher-subscriber systems, distributed systems, fault tolerance and checkpointing. In this section, we present some closely related works and present novel research contribution in context to existing works. We categorize the related work in two major areas: *Push-based communication* and *Checkpointing*

2.1.1 Push-based communication

Content Centric Networks follows a pull-based approach of communication as depicted in [21]. This puts a constraint on developers to model their applications complying with the presented architecture. Several distributed applications in a content centric network paradigm have been developed which provides a pull-based peer-to-peer communication [18, 20, 35, 37]. Jacobson et al. [20] presents a Voice-over IP application in a content-based paradigm. The authors utilize the *active naming* capability of CCN to actively request for content not yet produced by the producer and by recursively sending new Interest for every satisfied one to maintain the number of outstanding interests in the pipeline. Similarly, Zhu and Afanasyev [37] developed a distributed chat-based application working on a similar principle of satisfying outstanding synchronization interest with a data on state change. Gusev [18] have developed a real time audio-visual communication over NDN where users can discover each other from within the application and begin conferencing. The authors

have employed the use of different user namespaces to provide session initiation capabilities. Wang et al. [35] and Qu et al. [30] have also demonstrated peer-to-peer multiplayer games in CCN using a similar concept.

The existing distributed applications in IP networks follow a push-based approach [34]. A lot of research has been done in implementing push-based communication approach in Content Centric Networks. Bodapati et al. [9], Carzaniga et al. [12], Chen et al. [14, 15] have tried to implement push-based communication model in CCN's and use them as publish/subscribe architecture. The paper by Carzaniga et al. [12] and Chen et al. [14] developed a sender-initiated publish-subscribe scheme by employing new packet types to register address of subscriber. The authors, in their paper, also discuss the reasons why sending a non-cached interest to a node to initiate CCN communication is not favorable for a pub-sub network in CCN. The authors feel that as the scheme involves three-way communication rather than the usual two-way in CCN, it will impose high overhead in the system. We refute these claims made by the authors and show that three-way communication imposes a very nominal overhead but provides an optimal sender-driven communication in CCN which it currently lacks. Moreover, our design abstracts the sender-driven functionality on application layer [1] and does not require changes to CCN architectures, such as using new packet types other than Interest and Data.

2.1.2 Checkpointing

Checkpointing and rollback-recovery are well known techniques that allow processes to make progress in spite of failures [33]. Several algorithms have been designed and developed to take consistent checkpoint in a distributed network [11, 22, 24, 26]. Over the time, coordinated checkpointing algorithms have emerged as the most commonly used type of algorithms to produce consistent checkpoints with lowest system overhead [10, 16].

Using the above-mentioned checkpoint algorithms, several checkpointing packages [2] have been developed which can checkpoint single machine or distributed systems in TCP/IP network based applications [23, 32]. DMTCP, developed by Ansel et al. [6], is a user-level, transparent checkpoint-restart tool which can checkpoint distributed multi-threaded as well as desktop applications. Moreover, DMTCP allows developers to add functionality to through the use of plugins which other checkpoint packages fail to provide. We thus base all our implementation on DMTCP.

[1] This is an advantage as applications of different nature can exploit this technique according to their required purpose

[2] These checkpointing packages are both kernel-level and user-level in nature

To the best of our knowledge, their is no prior work/package which aims to checkpoint applications in Content Centric Networks. Our work, in subsequent sections, proposes to fill the gap.

2.2 Research Contributions

In context to the closely related works, this report makes the following research contributions:

1. Designing an efficient sender-initiated application-layer based algorithm for distributed systems in CCN;
2. Implementing the designed algorithms in CCNx and exporting them as C++ library functions to be used by developers.
3. Studying the various failure conditions of exiting algorithms for checkpointing in CCN.
4. Designing and implementing novel checkpointing algorithms for CCN-based applications as an overlay on existing algorithms and software.
5. Evaluating the performance of above-mentioned system compared to earlier works in this area.

Chapter 3

Background

3.1 Content Centric Networking

The current TCP/IP networks were built to solve age old problem of *resource sharing* which was prevalent in 70s. The communication model that resulted is a conversation between two machines, one wishing to use resource and other providing access to it. The IP packet thus contains two identifiers, one to specify the source address and other to specify the destination. In a TCP/IP network, all nodes are connected by such two-way TCP conversations.

However, as the time progressed, the size of data to be transferred between nodes have increased to scale of *exabytes* $(10^{16} bytes)$[3]. This proves that the users of the network have become more interested to share data rather than resources amongst machines. To better accommodate the emerging pattern of internet based on content than location, lot of significant work has been done by researchers in this direction [5, 17, 27, 36] . The most formal approach to content-oriented communication was provided by Jacobson et al. [21] which was named *Content Centric Networks* ***(CCN)***.

CCN intends to provide a substantial degree of flexibility for users and end-systems to obtain information without regard to their location or source. The significant difference between IP and CCN is that the thin waist of TCP stack[1], as in today's IP architecture is the centerpiece of CCN (see figure 3.1). This marks a significant difference between operations of data delivery between IP and CCN. In ways of working, the sender and receiver nodes in current IP networks is replaced by *Consumer* and *Producer*. A Producer is a node which produces a data and Consumer is the node requesting that data from the producer. In this section, we will briefly cover basic concepts of CCN upon which rest of our work rests.

[1]Note that the transport layer of IP network is second layer in CCN stack. Thus, all the transport level functionality of IP networks is handled at lower levels in CCNs

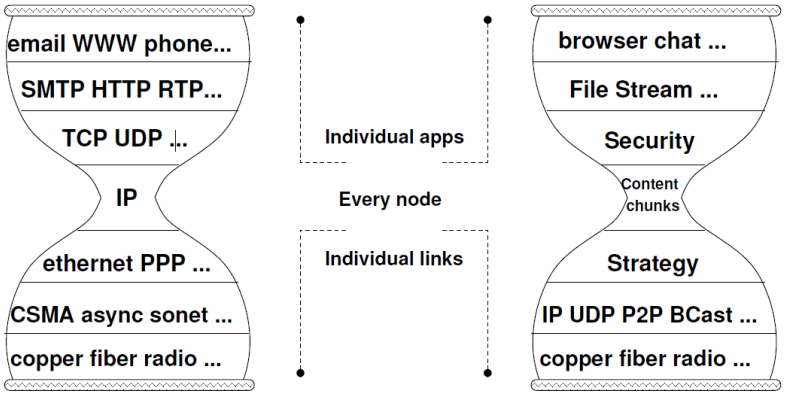

Fig. 3.1 Thin waist of network stack in CCN as proposed by Jacobson et. al. [21]

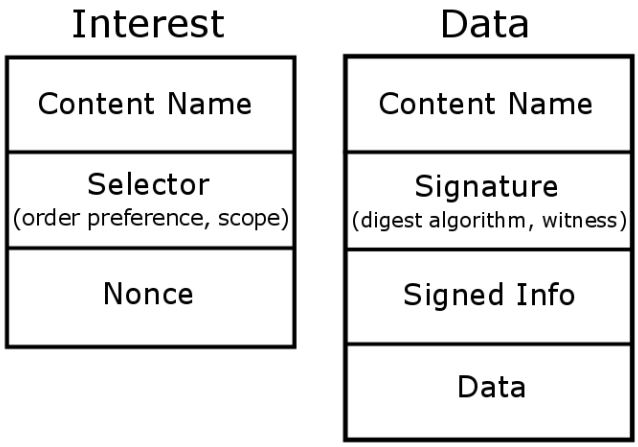

Fig. 3.2 CCN Packet Types

3.1.1 CCN Packet Types

There are two basic packet types in CCN, namely *Interest* and *Data*. The structure of both packets has been depicted in figure 3.2. The packet types have been explained in details below:

1. **Interest Packet:** The Interest packet is sent when a consumer requests for a data packet. It is composed of the following fields:

   - *Name*: This field indicates the name of the desired data produced by the producer and requested by the consumer.
   - *Selectors*: Provides the order preference and scope which specifies the preferences and other restrictions if more than one Data packets can satisfy the Interest.

- *Nonce*: A random nonce value which is used to prevent Interest looping in a network.

2. **Data Packet**: The data packet is produced by the producer and contains the data in which the consumer has an *"interest"*. The data packet is made up of the following fields:

   - *Name*: A Data packet can be used to satisfy the Interest as long as the name of the data packet is a prefix of that of the incoming Interest packet. Naming in CCN has been discussed in details in later sections of this chapter.
   - *Content*: The actual content requested.
   - *Signed Info*: This field contains additional information such as publisher's key name, timestamp etc.
   - *Signature*: This field contains publisher's signature. Unlike IP networks, CCN encrypts the data rather than network path/ links.

3.1.2 Hierarchical Names

Every chunk of data in CCN is named using hierarchical naming structures termed *components*. Names in CCN are human readable in nature, that is, they are specified in group of understandable language formats. The hierarchical structure of a name should start with an organization name, followed by a tree-like structure scoping. For example, a possible data name to satisfy a CV in a CCN is `/iiitd.ac.in/MUC/nitinder_cv.pdf`, where **/** divides the CCN naming components. The name above can be shown as a tree depicted in figure 3.3.

Such a hierarchy is useful in allowing applications to represent relations between data pieces. It also enables name-based routing to scale in a network. This is because such a naming hierarchy facilitated *"scoping"*, that is, some name may only be meaningful to a specific network domain and are not meant to leak to other domains. For example, `/iiitd/computerscience` prefix indicates that the interest for the names under it should only be broadcasted in the computer science department of IIITD and should not be forwarded beyond that boundary. On the other hand, there may be some names which are globally unique in nature and can be specified by any application in the network. The name depicted in figure 3.3 is a good example of such a globally unique name.

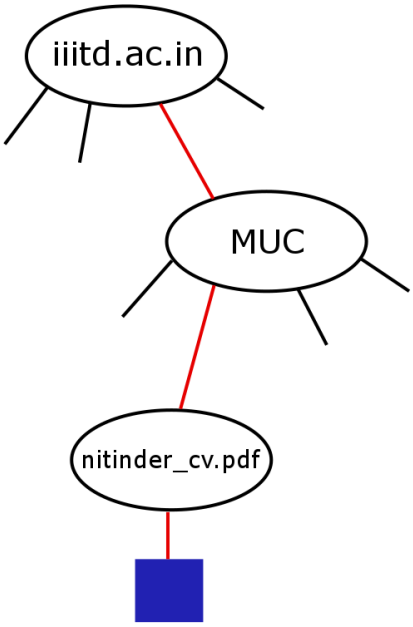

Fig. 3.3 Name Tree Hierarchy

3.1.3 Receiver-driven Communication

CCN follows a receiver driven or pull-based communication paradigm. To receive a data packet, the consumer must send an Interest with the desired data name in the network. This packet reaches the in-between routers [2] on a particular face [3] which the router keeps note of. A CCN router consists of following three components:

1. **Forwarding Information Base (FIB):** FIB is used to forward Interest packets towards potential producers of matching Data. The FIB is populated by a *name-based routing protocol*. The next router in line routes this packet to the next appropriate face until it reaches the desired node. The FIB follows a learning routing protocol which enables it to learn which faces satisfies which organization component.

2. **Content Store (CS):** This is same as buffer memory present inside the router. All answered data packets routed through a router are stored in the Content Buffer for later use. Whenever a new interest for a data arrives at a face, the router checks Content Store for taht particular data. If the data is found locally, the router replies to

[2]The only difference between a CCN and an IP router is that a CCN router is cache-enabled and can store the in-network data for future use.

[3]The *interface* is termed as a face in CCN as packets are forwarded on network interfaces but also exchanged between applications running atop of CCN.

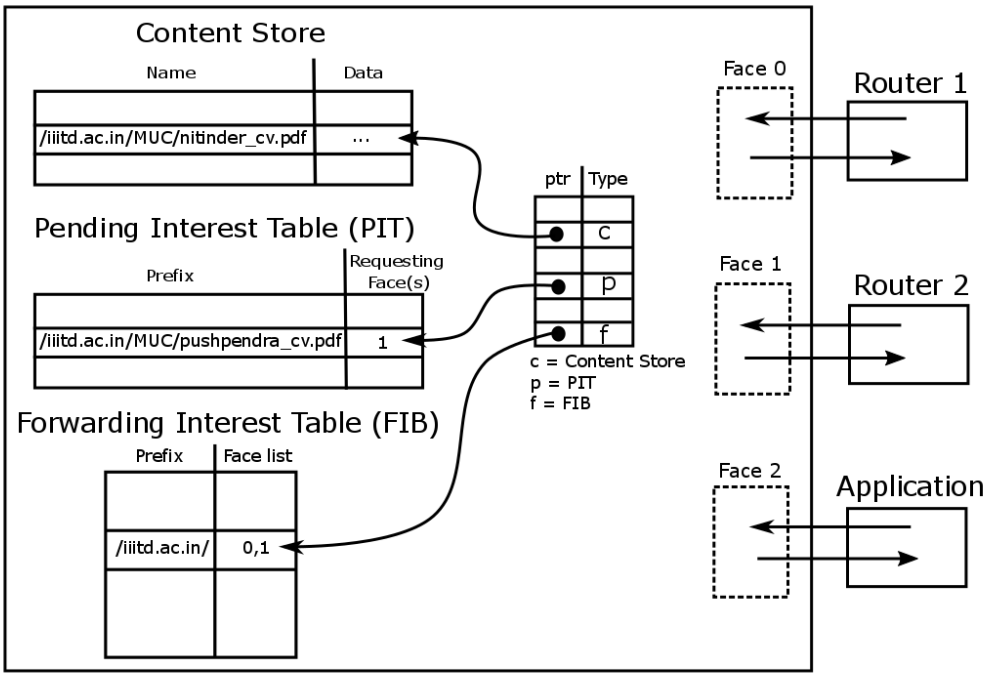

Fig. 3.4 CCN-based Router Structure

the interest; thus saving communication cost of re-routing the same interest back to producer.

3. **Pending Interest Table (PIT):** The interests that are yet to be satisfied are stored in PIT of router. When multiple Interests for same data are issued by different/same consumer(s), the router only forwards the **first** (and one) Interest towards producer and marks that Interest in PIT. When the Data arrives back at the router, it looks up the PIT, finds the matching entry and forwards the Data to all interfaces listed. An entry at PIT can be removed only if the router receives the Data packet or PIT entry reaches its time limit.

Due to the intelligent data plane routing of CCN, it naturally supports multicast data delivery. Moreover, the per-packet state allows routers to monitor packet delivery performance of different interfaces. Due to such a design, Content Centric Networks are also *Disruption Tolerant Networks (DTN)* in nature.

3.2 Checkpointing

Checkpointing is an an important process which stores the consistent state of a process on a stable storage, from which the process can resume its computation after a failure. Checkpointing enables distributed applications to make progress despite of transient failures. The failures under consideration are hardware errors and transaction aborts, i.e., those that are unlikely to recur after a process restarts. Several checkpointing-recovery techniques have been extensively designed in the literature [11, 22, 26].

3.2.1 Types of Checkpoints

In a distributed system, the states of processes are dependent on one another due to inter-process communication. Due to this inter-process communication, the checkpoint take can be **consistent** or **inconsistent** in nature. The notion of the global state of the system was formalized by Chandy and Lamport [13] . The global state of the system after a checkpoint is considered **consistent** only if it satisfies following conditions:

1. No such message exists in the system which has been received by a process without being sent by another process.

2. No such message exists in the system which has been sent by a process but has not been received by the recipient process.

If a checkpoint doesn't satisfy the conditions above, it is considered as **inconsistent** in nature.

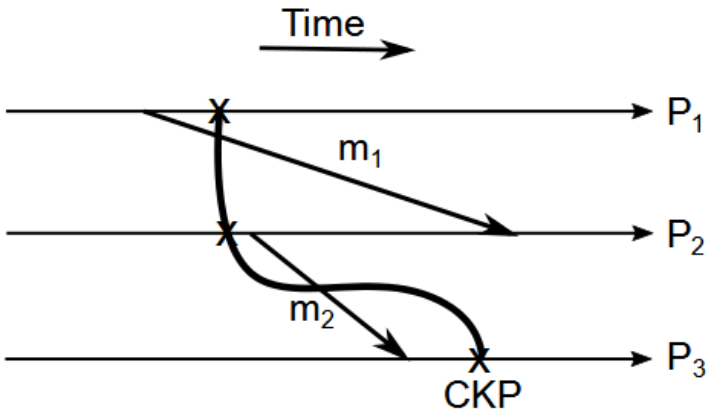

Fig. 3.5 Inconsistent Checkpoint

Figure 3.5 shows an inconsistent checkpoint scenario taken between communicating processes P_1, P_2 and P_3. As seen from the figure, process P_2 takes a checkpoint before sending message m_2 and process P_3 takes a checkpoint after receiving message m_2. This defies rule 1a of consistent checkpoint scenario. Similarly, process P_1 takes a checkpoint after sending message m_1 and process P_2 takes its checkpoint after receiving message m_1. In this case, the checkpoint taken fails to comply with the rule 1b.

If the checkpoint taken is inconsistent in nature, the rollback recovery after a failure on process p_i would require all the dependent processes on that process to rollback. As the checkpoint is *indeterministic* in nature, a rollback-recovery would lead to **domino effect** where all the dependent processes will keep rolling back and would restart its execution from initial computation state.

The problem of **domino effect** can be solved by taking a consistent checkpoint. For example, the inconsistent checkpoint scenario depicted in figure 3.5 can be transformed to a consistent checkpoint depicted in figure 3.6.

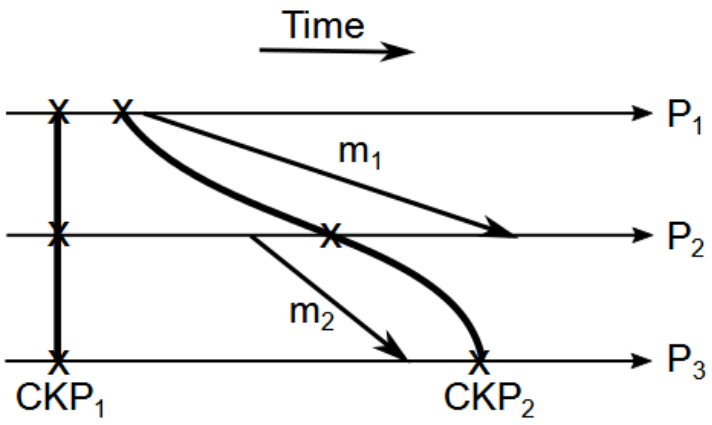

Fig. 3.6 Consistent Checkpoint

3.2.2 Checkpoint Algorithms

There are two different types of checkpoint algorithms:

1. **Independent Checkpointing Algorithm**:
   The independent approach saves its individual checkpoints periodically regardless of the dependencies with other processes. When a process recovers from a failure, a *consistent* set of checkpoints are selected for process affected by failure by exchanging dependency information [8] .

2. **Coordinated Checkpointing Algorithm**:
   The processes under coordinated approach coordinate their checkpointing actions such that each process saves only its most recent checkpoint, and the set of checkpoints in the system is guaranteed to be consistent. When a failure occurs, the system restarts from the most recent checkpoint.

A rollback-recovery from independent checkpoints results in *domino effect* which is the biggest disadvantage for this algorithm. As, this problem is absent in coordinated checkpointing, it is the most widely used checkpoint algorithm [11, 22].

3.3 Distributed Multi-Threaded Check-Pointing

Distributed Multi-Threaded Checkpoint (DMTCP) is a research-based, transparent, user-level checkpoint-restart package for distributed applications. DMTCP is able to checkpoint fork, exec, ssh, TCP/IP sockets, UNIX domain sockets, shared open file descriptors etc. DMTCP supports traditional cluster-based high performance distributed applications along with typical desktop applications.

DMTCP follows a *coordinated checkpointing algorithm* to ensure a global consistent state at each checkpoint. In coordinated checkpointing algorithm, all the processes and threads cluster-wide are simultaneously suspended during checkpointing. Network data on transmission link and in kernel buffers is flushed into the recipient process's memory and saved in its checkpoint image. After a checkpoint or restart, this network data is sent back to the original sender and re-transmitted prior to resuming user threads.

3.3.1 Architecture

A computation running under DMTCP consists of a centralized coordinator process and several user processes. These user processes can be local or distributed in nature. The communication between these user processes is through sockets, shared-memory etc. Moreover, every process communicates with the coordinator through a checkpoint thread. This thread is responsible for saving9 checkpoint images when requested by coordinator. Similarly during restart, it initiates the restart phase at each process and restarts the entire process along with its threads from the image. The thread is dormant during normal execution of the process and is only active during checkpoint/restart procedures.

The architecture of DMTCP is depicted in Figure 3.7.

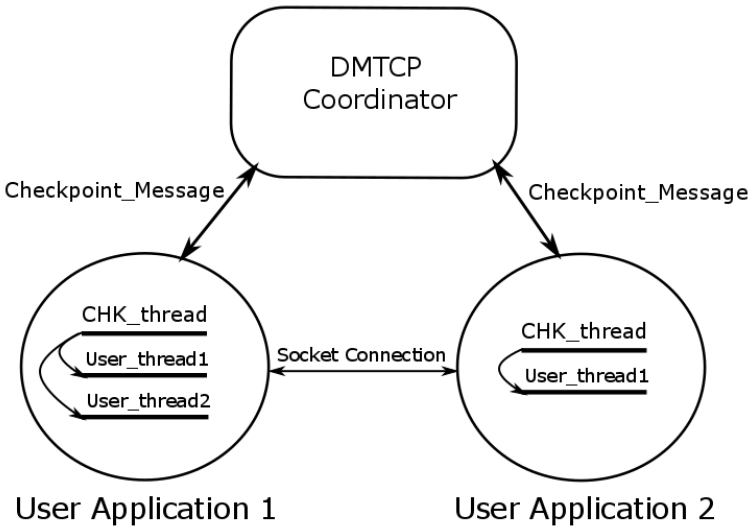

Fig. 3.7 DMTCP Architecture

3.3.2 Checkpointing Algorithm

On receiving the checkpoint request from the coordinator, the checkpoint signal is sent to all the user threads in the process by the checkpoint thread. This quiesces the user threads by forcing them to block inside a signal handler previously installed by DMTCP. The checkpoint image is created by writing all of user-space memory to a checkpoint image file. Each process has its own checkpoint image stored locally. Prior to creating the checkpoint image, the checkpoint thread also copies into the user-space memory, any kernel state that is required to restart the process such as the state of associated with network sockets, files, and pseudo-terminals.

DMTCP checkpoint algorithm is *blocking* in nature as it uses marker messages to flush data packets present in the sockets. The algorithm executes in stages where a barrier to next stage is released only if all the communicating nodes under coordinator have completed that stage. The algorithm used by DMTCP to checkpoint user processes is provided in Algorithm 1.

3.3.3 Restart Algorithm

After taking a successful checkpoint, the DMTCP groups all restart images from a single node under a sing `dmtcp_restart` process. Once all the images are connected back to the coordinator, the coordinator begins the process of restarting the application from the

Algorithm 1 DMTCP Checkpointing Algorithm

1: **Wait** for checkpoint process to start
2: **Suspend** all user threads
    Release Barrier: **Suspended**
3: **Leader Election** for each shared file descriptor between nodes
    Release Barrier: **Election completed**
4: **Drain** TCP sockets using *flush* packets
    Release Barrier: **Drained**
5: **Save** user-space memory to a checkpoint file
    Release Barrier: **Checkpointed**
6: **Refill** sockets with data
    Release Barrier: **Refilled**
7: **Resume** user threads
8: **Go to** 1

checkpoint. The algorithm used by DMTCP to restart an application from a checkpoint has been depicted in Algorithm 2

Algorithm 2 DMTCP Restart Algorithm

1: **Reopen** file descriptors
2: **Reconnect** sockets across the network
3: **Fork** user processes to same number prior to checkpoint
4: **Rearrange** file descriptors for each user process
5: **Restore** local process memory and thread
    Release Barrier: **Resume from Barrier 5 of Checkpoint Algorithm**
6: **Refill** sockets with data
    Release Barrier: **Refilled**
7: **Resume** user threads
8: **Go to** 1

3.3.4 Checkpoint Consistency

To initiate a checkpoint, the coordinator broadcasts a quiesce message to each process in the computation. On receiving the message, the check-point manager thread in each process quiesces the user threads, sends an acknowledgment to the coordinator, and waits for the *drain message*. After receiving acknowledgments from all processes, the coordinator lifts

the global barrier and broadcasts the drain message. On receiving the drain message, the checkpoint manager thread sends a special cookie, that is *marker message* through the "send" end of each socket. Next, it reads data from the "receive" end of each socket until the special cookie is received. Since user threads in all the processes have already been quiesced, there can be no more in-flight data. The received in-flight data has now been copied into user-space memory, and will be included in the checkpoint image.

On restart, the peer processes refill the network buffers, by pushing the data back into the network through the "send" end of each restored socket connection. The checkpoint manager thread then sends a message to the coordinator to indicate the end of the refill phase and waits for the *resume* message. Once the coordinator has received messages indicating end of refill phase from all involved processes, it lifts the global barrier and broadcasts the resume message. On receiving the resume message, the checkpoint manager un-quiesces the user threads and they resume executing user code.

Chapter 4

Distributed Applications in CCN's

As Content Centric Networks are designed as *Receiver-driven* or *Pull-based* in nature, the distributed applications in CCN needs to cater to such communication paradigm to function [20, 35, 37]. Moreover, as the content is produced by producer at runtime, the consumers do not know about the exact name of the content to request.

To tackle the above mentioned problem, the CCN's employ *active names* to request for data produced at run-time. The consumers of such data send interests with the name of data yet to be produced. When this Interest packet reaches the producer, the producer guesses the data required by consumer using the name of the incoming interests, generates the data with the same name and satisfies the interest. This approach has been used by several applications in CCN [20, 37] which are able to send/receive data to/from remote nodes.

Even though this approach allows distributed processes to communicate with each other at runtime, it is still receiver-driven in nature. The existing paradigm which is extensively followed by distributed applications in TCP/IP based distributed applications is *Sender-driven* or *push-based* in nature. In Sender-driven applications the producer produces the data and sends it to desired consumer directly. Due to this mismatch between communication paradigms of IP and CCNs, a transformation of such application to new paradigm is required which would induce workload on developers of such applications. One major problem of using existing approach of active names is that it would require flooding the entire network of distributed processes with outstanding Interest packets. This would lead to high network contention[1]. Such a scenario has been depicted in figure 4.1 where I_n depicts the Interest packet generated by P_n to request different data using *active names* and D_n denotes the data packet generated by P_n.

[1] If there are 'n' different processes for an application, it would require sending $O(n^n)$ outstanding interests in the network.

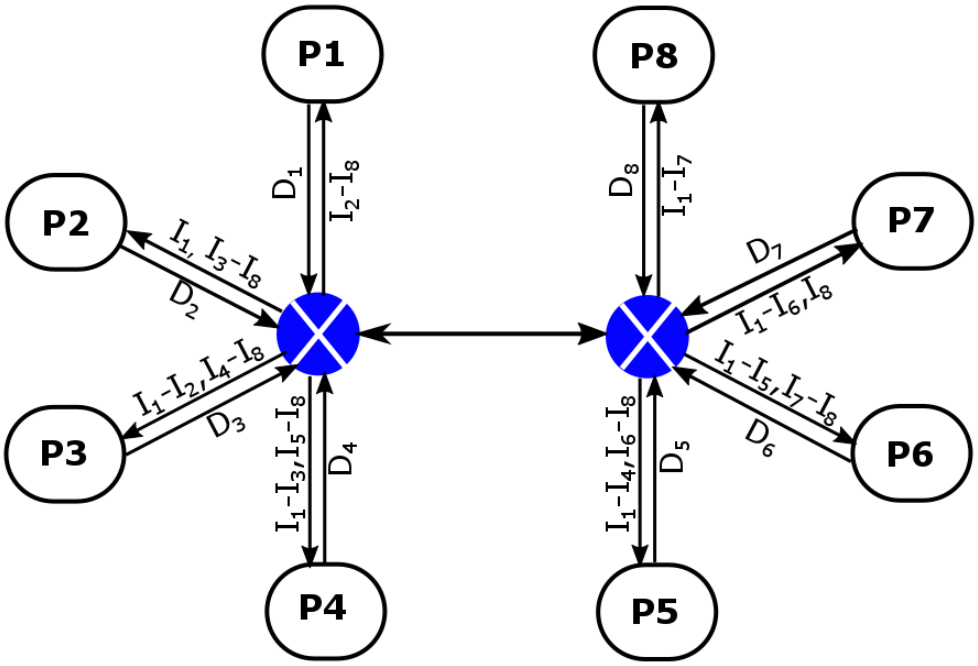

Fig. 4.1 Congestion while sending outstanding interests

We have thus come up with a novel approach to provide push-based communication paradigm for such distributed applications in CCN. Our solution enables producer to send the produced data to desired consumer at runtime using Interests and Data packets of CCN. One major advantage of using our approach is that no changes need to be done in the application source code apart from changing the **Send()** and **Receive()** TCP/IP socket functions to those of CCNs. We provide an API to developers using C++ libraries which can be directly imported in the application.

4.1 Communication Structure

The general communication pattern for applications in CCN is two-tier, that is, the consumer sends an Interest packet to which the Data packet containing the content requested is sent back. We extend this communication pattern to three-tier to transform the pull-based to push-based approach. Moreover, we use the unique hierarchical naming of CCN packets to perform a unique action. Our approach is guided by following **pre-assumptions**:

- Every node knows about names of other nodes running the same distributed application in the network.

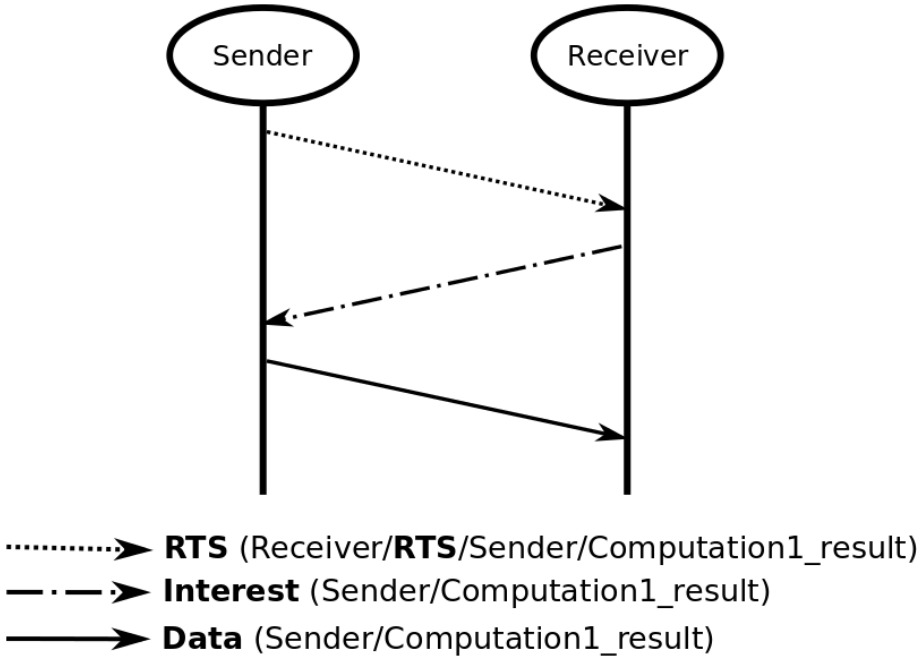

Fig. 4.2 Designed Communication Architecture

- Every node is defined by a unique name which is pre-appended by the application name the process is running.

To run a sender-driven distributed application in CCN we use an approach very similar to solving *hidden terminal problem in IEEE 802.11 networks*. We use an RTS Interest packet which acts as a notification to destination about an incoming data. The approach works in steps specified in algorithm below:

Algorithm 3 Sender-driven Communication Algorithm

1: *Producer* **produces** the intended data
2: *Producer* sends **Request-to-Send (RTS)** notification Interest packet to desired *Consumer*
3: *Consumer* **extracts** the name of data from RTS and sends the interest for the data produced
4: *Producer* **receives** the Interest packet and satisfies it with data
5: *Consumer* **receiver** data sent by *Producer*
6: **Go to** 1

Figure 4.2 shows the designed sender-driven communication. More details have been discussed in later sections.

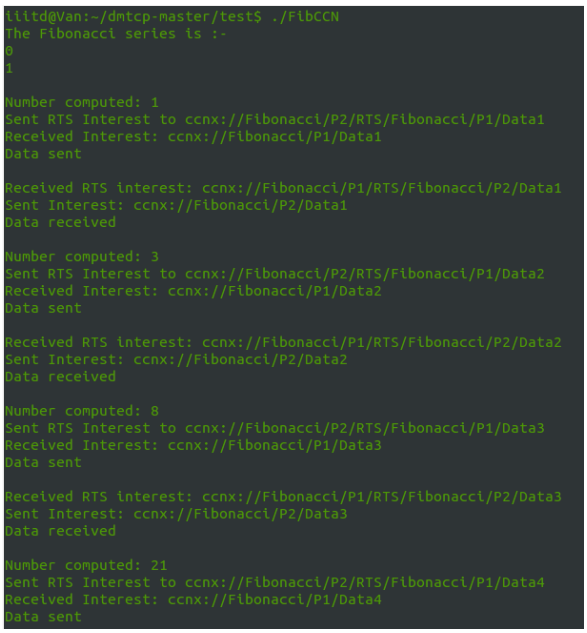

Fig. 4.3 Screenshot of CCN Fibonacci calculator using proposed algorithm on Process 1

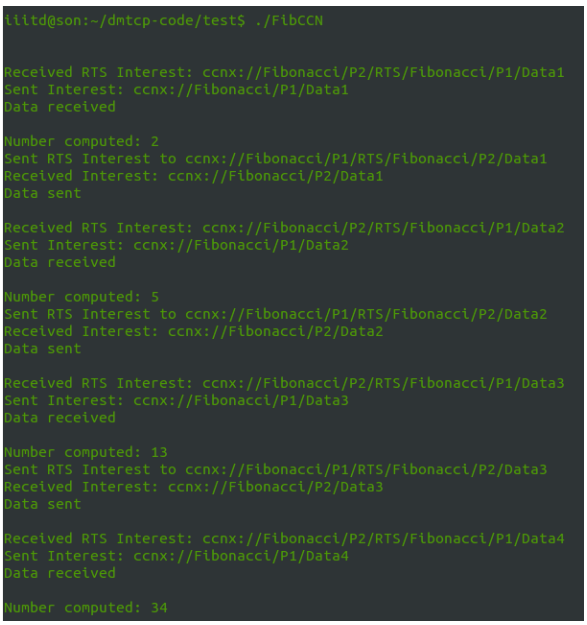

Fig. 4.4 Screenshot of CCN Fibonacci calculator using proposed algorithm on Process 2

4.2 Packet Structure

The designed communication approach utilizes the naming capability of CCN packets to uniquely identify the packet type and function. Several applications in CCN utilizes Interest packets with ***active names*** to request for data packets not yet been produced by producer.

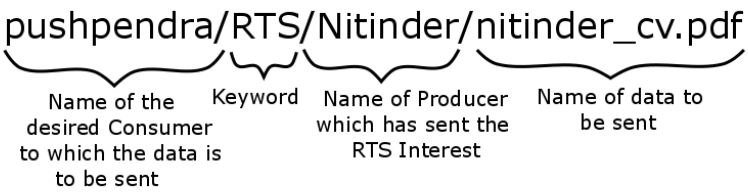

Fig. 4.5 Request-To-Send Interest Packet Naming Structure

Using this approach, we design ***pro-active names*** which uses the name of the Interest to specify the *function performed* by the process. The packets with pro-active names is similar to the control message in TCP networks. The Interest packets with pro-active naming is assumed not to be cached in the router and thus ensure end-to-end delivery. In this section, we are going to discuss the packet structures using pro-active name concept for packets utilized in our approach.

4.2.1 Request-to-Send (RTS)

Request-to-Send or an RTS packet is key to push-based communication approach in CCN's. An RTS packet notifies the consumer about a pending data to be sent and also the name of the data which has been produced. Whenever the consumer receives an Interest packet, it checks the packet name for keyword **RTS**. If the keyword is found in the name, the Consumer extracts name of the data post-pended after `RTS` keyword and sends a regular Interest packet with extracted data name.Various aspects about the RTS packet have been discussed below:

RTS Structure

The RTS Interest packet is same as any Interest packet in Content Centric Networks. The only difference between an RTS and an Interest packet in CCN network is that the RTS packet is not cached in any in-network routers in their Pending Interest Table's. This essentially means that producer is not expecting a reply for this Interest packet and this packet will only serve as a notification to Consumer node. In CCNx emulator [2], this can be achieved by setting the `Interest_Lifetime` value as **0**.

RTS Naming

An RTS packet name sent from producer node to consumer node is made up of four components:

[2]Read CCNx Technical documentation [4]

1. **Consumer Name**: This denotes the destination node name to which RTS is to be sent. The destination naming follows the same naming convention as that of regular CCN packets. For example, in the example shown in figure 4.5, the RTS is being sent to `pushpendra` which follows no name scoping and is registered in the network as a singular node without any organization name. If `pushpendra` were to be part of an organization, say `iiitd.ac.in/MUC`, then the consumer name of the RT packet would be denoted as `iiitd.ac.in/MUC/pushpendra`.

2. **RTS Keyword**: The consumer name is followed by a `RTS` keyword which distinguishes the RTS packet from general Interest packets in CCN. On receiving this packet, the consumer node checks the URI name of the packet for `RTS` keyword. The name components after `RTS` keyword is used to form the Interest to be sent back to Producer.

3. **Producer Name**: The `ProducerName` is used to identify the producer which has sent the RTS Interest. As CCN is designed to send/receive data and routing is done through PIT entries at in-network routers, CCN provides no mechanism of knowing the name of source node of Interest packet. In our approach, the Consumer needs to determine the node which has sent the RTS such that the Interest packet for the data can be routed back to that same node. Due to this, the Producer source name is made part of the RTS Interest.

4. **Content Name**: The final component of the RTS interest is the content name produced by the producer which it needs to push to the consumer.

4.2.2 Interest Packet

After the receipt of `RTS` packet, regular CCN communication follows between Consumer and Producer. The Interest packet sent is regular Interest packet which is sent back from Consumer to Producer after the Consumer receives the RTS packet sent by the Producer.

Interest Structure

The Interest packet sent is same as regular Interest in a CCN. Detailed structure of Interest packet and its components have been discussed in Section 3.1.1.

Interest Naming

The Interest packet sent back to Producer by Consumer has sole purpose of requesting for Data packet from Producer.The name of the Interest packet would be the components

following the `RTS` keyword in RTS Interest packet. For example, if Consumer receives a RTS Interest packet with name depicted in figure 4.5. In this case, the name of the Interest packet would be:

```
Nitinder/nitinder_cv.pdf
```

The Interest packet would be routed to producer `Nitinder` and request for data it specified in RTS Interest packet. In this case, the data packet requested is `nitinder_cv.pdf`

4.2.3 Data Packet

The data packet is sent by the producer after it receives the Interest from Consumer. General rules of Data packet are applicable. More information on Data packets and its structure is available in Section 3.1.1.

Chapter 5

Checkpointing in Content Centric Networks

The checkpointing in TCP/IP networks and algorithms used to checkpoint applications in these networks have been discussed in details in Section 3.2 and Section 3.3. However, a process in CCN is different from a process in regular TCP/IP networks. This is so because processes in regular TCP/IP networks follow **one-step communication**, that is, the sender node directly sends the data to the receiver node in a single step. However, processes in CCN follow **two-step communication**, that is, for every data packet to be sent in the network, an Interest packet needs to be relayed to the producer.

As the existing checkpointing algorithms have been developed keeping in mind regular sender-driven communication [11, 22, 26], these cannot be used for receiver-driven CCN architecture. Even if we use our designed approach, discussed in Chapter 4, to transform communication architecture of CCNs from receiver-driven to sender-driven, the communication becomes three-step in nature (RTS->Interest->Data). Ultimately, to solve this problem a new algorithm needs to be designed which would work for processes in CCNs.

5.1 Algorithms of Checkpointing in CCN

5.1.1 Independent Checkpointing

We will first discuss the possibility of taking Independent Checkpoint in a typical CCN communication. A typical CCN communication is a *single-step communication* which consists of an Interest packet sent by Consumer to which Producer replies with a Data packet. A checkpoint can be taken by consumer and producer without any consensus. This can lead to **nine** distinct checkpoint instances which is depicted in figure 5.1.

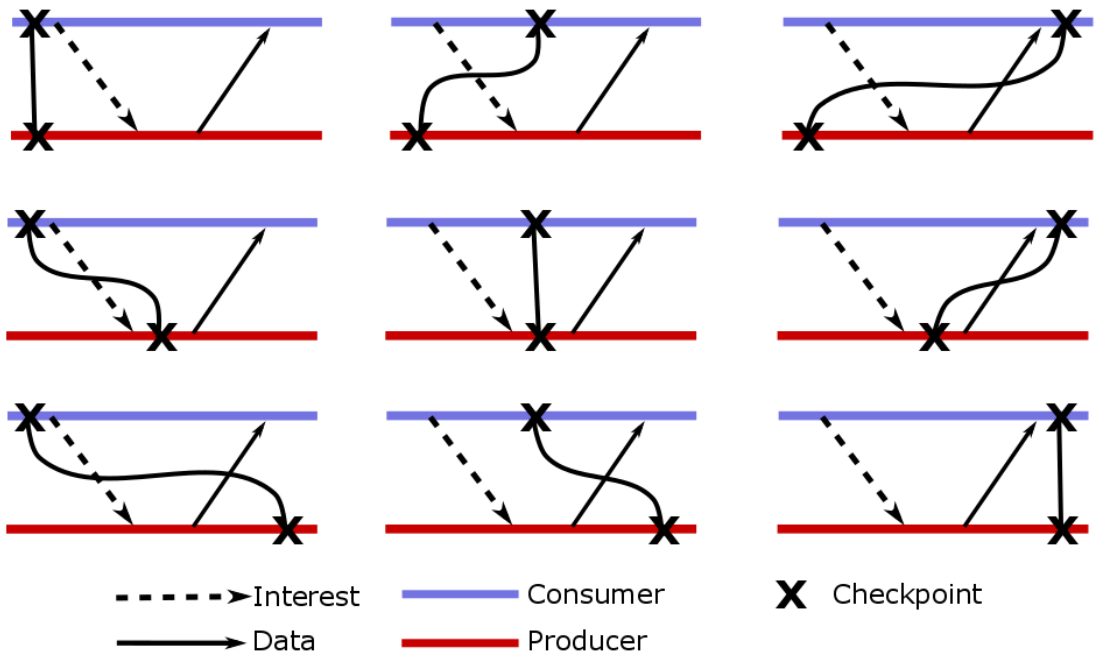

Fig. 5.1 Possible Checkpoint Scenarios in CCN

Group / Case	(a)	(b)	(c)
1.	Consistent	Inconsistent	Inconsistent
2.	Consistent	Inconsistent	Inconsistent
3.	Inconsistent	Inconsistent	Consistent

Table 5.1 Consistency of checkpoints in Fig 5.1

Many of these checkpoint instances are **inconsistent** in nature. This information has been provided in table 5.1 given below.

The detailed explanations of these scenarios have been depicted in next section.

5.1.2 CCN Checkpoint Consistency Rules

Figure 5.1 shows several independent checkpointing scenarios present in CCN. However, many of these checkpoints lead to an inconsistent checkpoint-restart for applications. We will now formulate consistency rules for checkpointing in CCN by extensively looking for failure conditions for scenarios described in figure 5.1.

1. **Data Consistency Rules**:
   According to existing checkpoint-restart algorithm discussed in section 5.1.2, a consis-

tent checkpoint must ensure capturing both "send" and "receive" events for each and every data packet sent in the network. As in CCN's, a *Data* packet is the central entity of the communication, the derived checkpoint consistency rules for *Data* can be as follows.

(a) *If a checkpoint saves the "receive" event of a Data, it must also contain the "send" event of that Data.*

(b) *If a checkpoint saves the "send" event of Data, it must also save the "receive" event of that Data.*

In case 2(c) shown in figure 5.1, the consumer takes its checkpoint **after** receiving the *Data* packet whereas the producer takes its checkpoint **before** sending that *Data*. As this scenario does not agree with the rules stated above, the resulting checkpoint is inconsistent. Case 1(c), 3(a) and 3(b) also fail to comply with the Data consistency rules mentioned above.

2. **Interest Consistency Rules**:
   In CCN, an *Interest* packet is issued to request *Data* from the producer and does not carry any content. However, consider a scenario where the "send" event of the *Interest* at the consumer is not a part of the checkpoint image. Restarting from this checkpoint would anyways force the consumer to resend the *Interest* to the producer regardless whether the "receive" event at producer is part of checkpoint or not. On the other hand, if the checkpoint contains the "send" event and not the "receive" event, the application would result in a deadlock after a restart. This is because the consumer is waiting for the *Data* to arrive whereas the producer is waiting for an *Interest* for the *Data*. Formally, CCN's *Interest* consistency rules states that:

   (a) *If a checkpoint contains the "send" event of an Interest, it must also store the "receive" event for that Interest.*

   (b) *If a checkpoint does not contain the "send" event of an Interest, it may/may not store the "receive" event of that Interest.*

   Case 1(b) shown in figure 5.1 fails to satisfy the rule (a) resulting which the checkpoint is inconsistent. It should be noted that case 2(a) satisfies condition (b) due to which the checkpoint taken in this scenario is consistent.tency

3. **Pending Interest Table (PIT) Consistency rule**:

Case 2(b) depicted in figure 5.1 is an interesting case to note as even though it satisfies both *Data* and *Interest* consistency rules mentioned above, yet the checkpoint taken is inconsistent.

The inconsistency of this checkpoint is due to the routing protocol followed in CCN. As discussed in Section 3.1.3, routing in CCN is dependent on the PIT entry of in-network routers that serve as the breadcrumbs for delivery of *Data* packets. Considering case 2(b), the PIT entry for the requested *Data* is deleted after it is received by the consumer (and after taking the checkpoint). The application will result in a deadlock state if it restarts after *Data* "receive" event. This is because the consumer is waiting for the *Data* to arrive whereas producer cannot route the *Data* due to missing PIT entries. The CCN's PIT consistency rule avoids this scenario.

(a) *If a consumer has receipt for sending Interest for Data, it should also have the receipt of receiving the requested Data from producer.*

A checkpoint in CCN must satisfy all the rules mentioned above to ensure a consistent state on application restart. After considering all the rules, only case 1(a), 2(a) and 3(c) result in a consistent checkpoint in CCN.

5.2 CCN Application Checkpoint (CCNAC) tool

From section 5.1.1, we establish that independent checkpointing algorithm cannot be used for taking consistent checkpoints in CCN. We thus develop **CCNAC**, which is a plugin for DMTCP [6] to checkpoint applications in Content Centric Networks. In subsequent subsection, we will provide detailed information regarding working of CCNAC.

5.2.1 Prerequisites

CCNAC checkpoints processes/applications communicating in Content Centric Networks. However, we assume the following points for running processes under CCNAC. Formally, a distributed system running on CCNAC works on following model:

1. Every node knows about other nodes running the same distributed application.

2. Every node is defined by a unique name which is pre-appended by the application name the process is running.

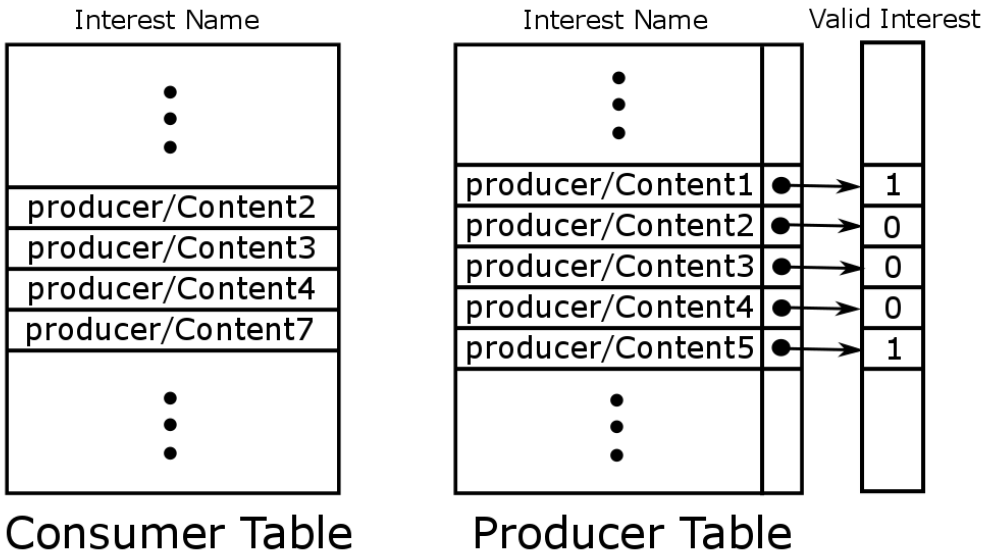

Fig. 5.2 CCNAC Data Structures

3. Each process can be defined by a CCN URI which is unique in the subset of that particular application. Moreover, each process knows the CCN name as well as the TCP address of the processes at the start of the application.

Moreover, CCNAC requires some extra data-structures to be a part of these applications for them to function properly.

1. **Consumer Table**

   As illustrated in figure 5.2, a *Consumer Table* is an array-like data structure present at consumer-side of the application. The Consumer Table contains the names of those pending applications which have been sent by the consumer but for which data packets have not been received. A simple algorithm to fill Consumer Table entries has been provided below.

Algorithm 4 Consumer Table Algorithm

1: ***Interest Sent***: **Add** Interest Name to Consumer Table
2: ***Data Received***: **Remove** Interest Name from Consumer Table

   The consumer Table is used by CCNAC to clear the communication channel during checkpointing. The details have been explained in later sections.

2. **Producer Table**

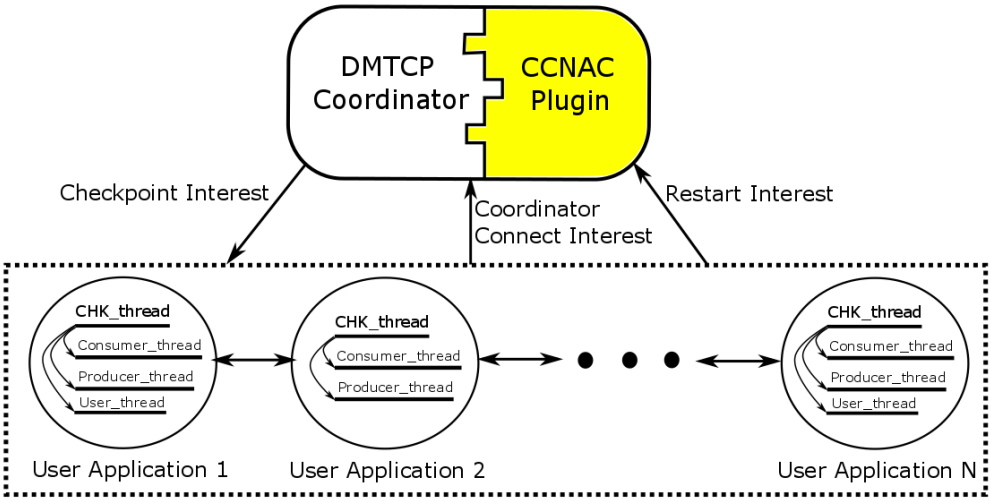

Fig. 5.3 CCNAC Architecture

A *Producer Table* is a producer-side, array-like table in which each table entry has a pointer which points to a binary field present in Valid Interest table. The table entry contains name of the interest received by producer. The binary value associated to an interest name denotes whether the Interest has been satisfied (this is, the data packet requested for that Interest has been sent) or not. An Interest has been fulfilled if Valid Interest entry is `1` and is unsatisfied is Valid Interest entry is `0`.

When a producer receives an Interest packet yet to be satisfied, it adds the packet to Producer Table and marks the associated Valid Interest entry as `0`. When the producer answers this Interest, it changes the Valid Interest value to `1`.

The Producer Table can also be updated by a `FLUSH` packet which is sent during checkpointing of applications in CCNAC. More on that will be given in details in later sections.

The Producer Table entries are used to validate while checkpointing that all the Interest received by the producer has been answered such that the checkpoint taken is consistent in nature.

5.2.2 Architecture

CCNAC is a plugin of DMTCP software (discussed in section 3.3) which enables checkpointing in Content Centric Networks. CCNAC follows a coordinated checkpointing algorithm by

performing blocking of packets on all communication channels. The architecture has been represented in figure 5.3. CCNAC is composed of following components:

1. **Coordinator**:
   CCNAC plugin controls existing DMTCP coordinator to coordinate between processes in a Content Centric Network. As CCN works on low-level TCP and UDP connections, CCNAC enables DMTCP coordinator to understand and detect CCN Interest and Data packets. On starting the coordinator, CCNAC provides a CCN URI to the coordinator which is unique amongst the group of distributed processes. CCNAC coordinator URI is composed of following parts:

   `OrganizationName/ApplicationName/Coordinator`

   For example, if CCNAC is initialized to checkpoint group of processes doing Fibonacci computation deployed in IIITD, MUC department, CCNAC would provide it CCN URI as:

   `IIITD/MUC/Fibonacci/Coordinator`

   The CCN URI can be automatically generated by CCAC on coordinator initialization or can be explicitly provided by the user. However, this unique URI should be known to all processes willing to register with the coordinator to checkpoint. Checkpoint procedure can be initiated by the coordinator on an explicit request from the user through its interactive interface or on expiration of a predefined checkpoint interval. It should be noted that the coordinator is a single point of failure since the entire computation relies on it.

2. **User Processes**:
   User Processes are general processes communicating in a CCN environment. These processes can either be distributed/centralized in nature. A user process under CCNAC will have a *Checkpoint_thread* executing which remains dormant unless it receives a checkpoint request from the coordinator. The checkpoint_thread is responsible for quiescing other threads active in the process and take a local checkpoint. Other than the checkpoint thread, a user process may contain a *consumer thread*, which is for sending Interest packets to other processes; a *producer thread*, which is responsible for registering incoming Interest packets and replying to them with appropriate data packets; and some other *user threads* which may be used to compute certain computation tasks at the process. Figure 5.3 shows two such user processes.

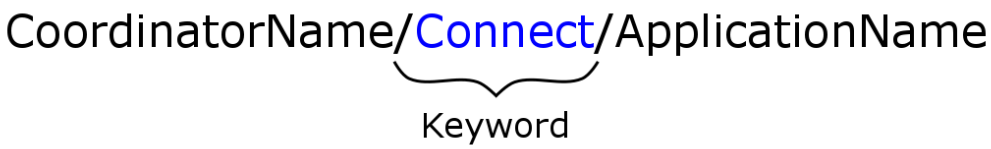

Fig. 5.4 Coordinator Connect Interest

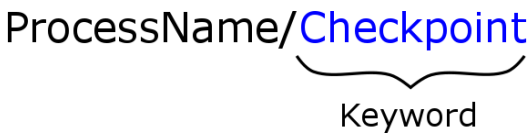

Fig. 5.5 Checkpoint Interest

5.2.3 Packet Structures

To take consistent checkpoint using CCNAC, we use Interest packets similar to RTS Interest used for enable push-based communication in CCN (Refer to Section). These Interest messages use pro-active names to initiate several actions taken by the process. Similar to RTS Interest message, these Interest messages are not stored in PIT entries of in-network routers and so they are not satisfied by receiving process.

1. **Coordinator Connect**:

   A checkpoint will be consistent in nature only if all the nodes of the application take a consistent checkpoint. In DMTCP, the nodes register with coordinator by providing their TCP address. As CCNAC is an overlay on existing DMTCP software, a coordinator connect Interest message is sent by all processes which are participating in the application to the coordinator. The naming structure of coordinator connect Interest is shown in figure 5.4.

   The coordinator connect Interest name is post appended with the application CCN name registering with the coordinator. The coordinator stores the name of application in its local cache. These names are later used by the coordinator to send checkpoint requests. For example, if the coordinator in a Fibonacci based application is named `Fibonacci/Coordinator` and the application trying to connect to it is `Fibonacci/ProcessA`, then the connect Interest name would look like:

   `Fibonacci/Coordinator/Connect/Fibonacci/ProcessA`

2. **Checkpoint**:

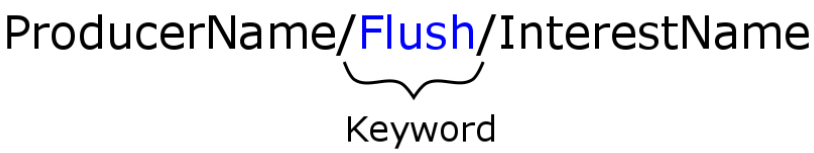

Fig. 5.6 Flush Interest

A *Checkpoint Interest* message is sent by the coordinator to all the processes registered with it for checkpointing. The checkpoint interest is an overlay over earlier used `CHK_msg` used by DMTCP coordinator in TCP/IP networks. The checkpoint manager thread at each user process sends the checkpoint signal to all the user threads in the process. This quiesces the user threads by forcing them to block inside a signal handler previously installed by DMTCP.

The naming format of the Checkpoint Interest is representated in figure 5.5. For example, if the process name is `Fibonacci/ProcessA`, then the checkpoint Interest message sent by coordinator to process would have name as:

`Fibonacci/ProcessA/Checkpoint`

3. **Flush**:

   To take consistent checkpoints, DMTCP employs a *"flush"* packet which clears all socket channels at the time of checkpoint. However, as shown in section [cite ccn checkpoint section], checkpointing an application in CCN requires satisfying all consistency rules involving receipt of Interest and Data packets, a TCP flush packet cannot be initialized at any time. Moreover, as CCN packets follows out-of-order delivery, the Interest and Data packets need to be synchronized before initiating checkpointing. A `Flush` Interest message is sent by a user process to all its dependent processes at the time of checkpoint to notify them of the unsatisfied Interest packets. The checkpoint thread at a user process does not initiate checkpoint until the Interest mentioned in Flush packet has not been satisfied.

   The format of the Flush Interest has been delineated in figure 5.6. For example, if `ProcessB` is yet to satisfy `ProcessA` interest `segment23`, the Flush Interest sent would be:

   `Fibonacci/ProcessB/Flush/segment23`

   Also, if there are no unsatisfied interests at the time of checkpoint, the process sends a `Flush/NULL` packet to dependent processes. The detailed algorithm has been discussed in later sections.

Fig. 5.7 Restart Interest

```
liltd@Van:~/dmtcp-master$ dmtcp_restart ckpt_FibCCN_90437f-43000-552ce370.dmtcp

1
Sent RTS Interest to ccnx://Fibonacci/P2/RTS/Fibonacci/P1/Data1
Received Interest
Interest received
data sent!
Received Interest

Received RTS interest
Sent CTS command Interest to ccnx://Fibonacci/P2/Data2
Received data
```

Fig. 5.8 Screenshot of restart from checkpoint

4. **Restart**:

   The Restart Interest message is sent by the user process to coordinator to restart from the checkpoint taken. DMTCP group all restart images from a single node under a single dmtcp_restart process. After receiving the Interest message, the process registers the name of the checkpoint taken with its checkpoint_manger thread which recreates all file descriptors. It then uses a discovery service to discover the new addresses for processes migrated to new hosts and restores network connections. It then forks a child process for each checkpoint image.

   Format of the Restart Interest is depicted in figure 5.7. For example, if the user process `ProcessA` wants to restart from its recent checkpoint, the Restart message sent would be:

   `Fibonacci/Coodinator/Restart/Fibonacci/ProcessA`

5.2.4 CCNAC Checkpoint Consistency Algorithm

DMTCP employs a strategy of “coordinated snapshots” using a global barrier. It follows a "blocking-type" coordinated checkpoint algorithm to ensure consistent checkpoints. As depicted in section [ccn rules], consistent checkpointing in CCN needs to ensure receipts of Interest and Data packets. It is to be noted from figure **??**, that Case 2(a), however produces a consistent checkpoint, can be ignored during checkpoint process due to computation

overhead. Therefore, only case 1(a) and 3(c) creates a consistent checkpoint in CCN-based applications.

CCNAC employs algorithms that ensure that any case should be reduced to Case 1(a) and 3(c) at checkpoint. As a user process communicating in a CCN environment may fulfill the roles of both producer and consumer, enforcement of algorithms for both these roles are necessary. The algorithms have been explained in details below:

Consumer-Side Algorithm

CCNAC employs a consumer-side algorithm which is implemented by each process which has sent an Interest message to other processes. The algorithm 5 formalizes the steps in the algorithm. The algorithm flow has been described in the in figure 5.9.

Algorithm 5 Consumer-side Checkpoint Algorithm

```
1:  Wait for checkpoint process to start
2:  if ConsumerTable ≠ NULL then
3:      for i ∈ index do
4:          Send Consumer/Flush/ConsumerTable[i]
5:      end for
6:      Wait for Data
7:      for i ∈ index do
8:          if Data == ConsumerTable[i] then
9:              ConsumerTable[i] ← NULL
10:         end if
11:     end for
12: else
13:     Send Consumer/Flush/NULL
14: end if
15: Process checkpoint stages in Algorithm 1
```

When a user process receives a checkpoint request from the coordinator, it first checks it Consumer Table for Interest message entries. A consumer table will have an Interest message name if the process has sent an Interest for a Data but it has not yet been satisfied. As taking a checkpoint at this stage would make it inconsistent, the process sends a flush message to the intended process for each entry in Consumer Table. When the data for the Interest arrives at the process, it deletes the respective Consumer Table entry and initiates further DMTCP checkpoint stages.

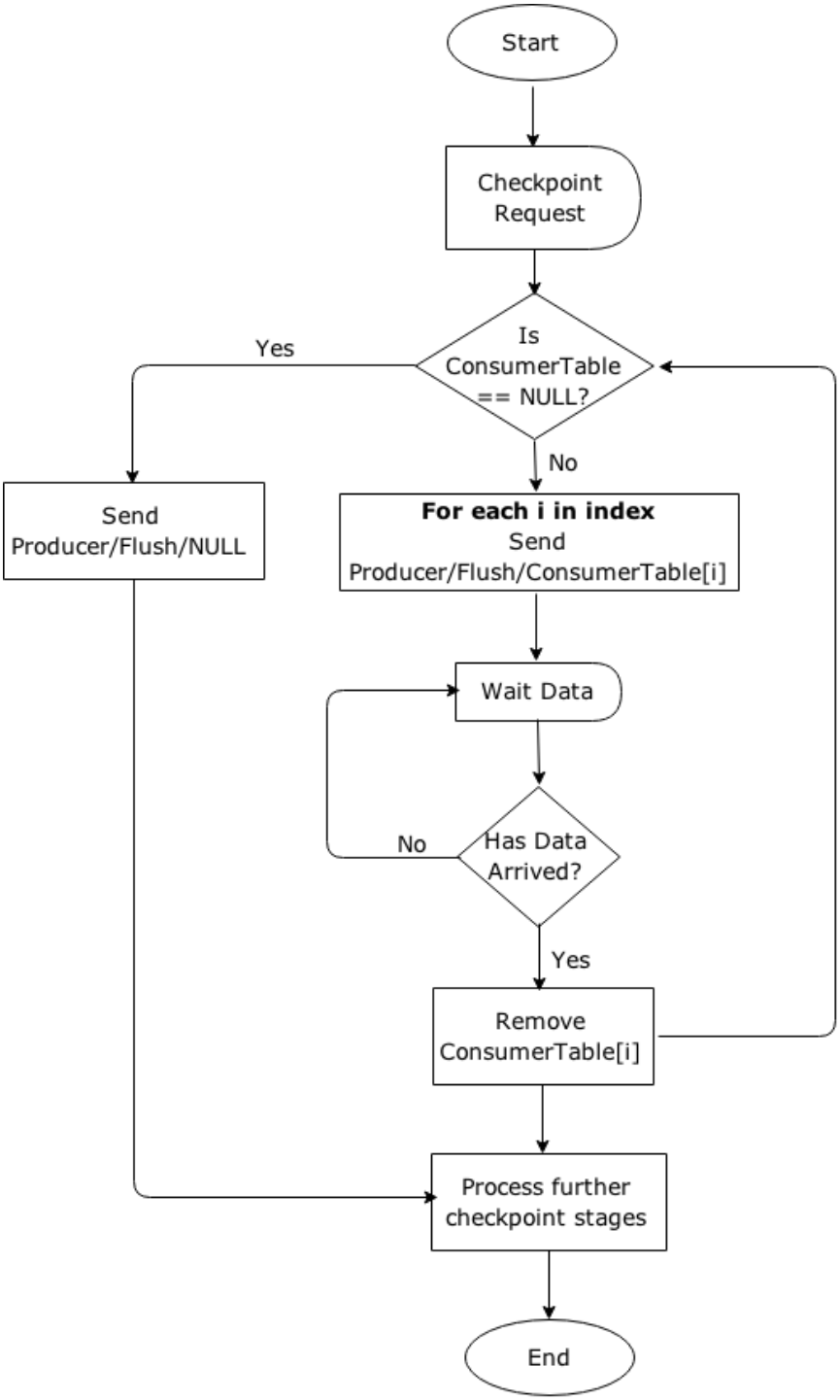

Fig. 5.9 CCNAC Consumer-side Algorithm

On the other hand, if Consumer Table is empty at the time of arrival of checkpoint request, the process sends a NULL flush packet to each dependent process and continues with further checkpoint stages. The algorithm terminates only when Consumer Table is free of any entries. It is important to note that Consumer table and its entries are not a part of the checkpoint and will be reinitialized at each process on restart.

Producer-Side Algorithm

CCNAC producer-side algorithm is implemented at each process which has sent a data to other processes in the network. The algorithm 6 formalizes the steps in the algorithm. The algorithm flow has been depicted in figure 5.10.

Algorithm 6 Producer-side Checkpoint Algorithm

```
1:  Wait for checkpoint process to start
2:  if IncomingFlush ≠ FLUSH/NULL then
3:      Name ← Namefrom(IncomingFlush)
4:      if Name ∈ ProducerTable[index] then
5:          if ValidInterest[index] == 1 then
6:              Go to 1
7:          else
8:              Wait for IncomingInterest
9:              if IncomingInterest == Name then
10:                 Send Reply
11:                 ValidInterest[index] ← 1
12:                 Go to 1
13:             end if
14:         end if
15:     else
16:         ProducerTable[index] ← Name
17:         ValidInterest[index] ← 0
18:         Go to 8
19:     end if
20: else
21:     Process checkpoint stages in Algorithm 1
22: end if
```

On receiving the checkpoint request from the coordinator, the process waits for a flush message from all the processes it is dependent upon in the network. If the incoming flush

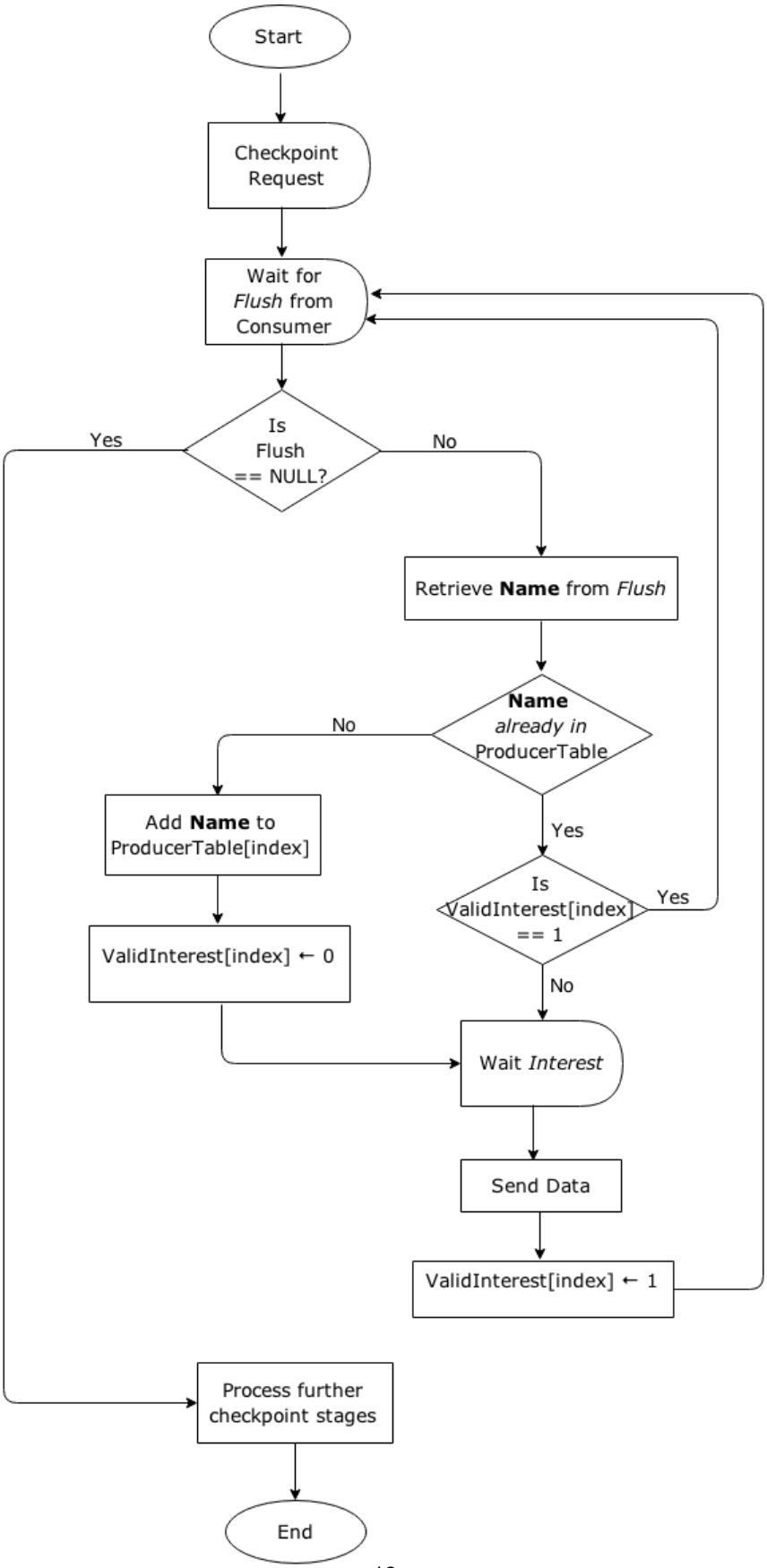

Fig. 5.10 CCNAC Producer-side Algorithm

message has an Interest name post appended to it, the process checks it Producer Table for that Interest entry. The Producer Table contains the name of Interest packet and the associated Valid field denotes whether the Interest has been satisfies by the process. If the Interest name is present in the table and is also valid, the process has satisfied this Interest before receiving the flush message. On the other hand, if the entry is invalid, the process is yet to receive the Interest message and it thus waits for the Interest packet to arrive. On arrival of the packet, the process satisfies the packet and marks the entry valid in the Producer Table.

If process has satisfied all the interest packets sent by other processes, it waits for NULL flush message from all processes involved. The algorithm terminates once all NULL flush messages have been received by the process. It is important to note that Producer table and its entries are not a part of the checkpoint image and are reinitialized on checkpoint restart.

Chapter 6

Evaluation and Analysis

6.1 Implementation

To evaluate the correctness of our approach, we implemented sender-driven communication approach and CCNAC plugin for DMTCP. The sender-driven communication functions were developed in C++ and were exported as libraries which can be used in any C/C++ CCN application. CCNAC plugin was developed on DMTCP v2.3.1 [6] and has been checked for compatibility with earlier versions DMTCP v2.3 and DMTCP v2.2.1. We used CCNx v0.8.2 [21] which is a software emulator developed by Palo Alto Research Center (PARC). The key component of CCNx is the *ccnd daemon*, which supports packet forwarding and caching.

The designed software was implemented on a testbed of 6 computers powered by Intel i5 - 2400 processors clocked at 3.10 GHz. All the computers have 4 GB of RAM storage and are interconnected with each other using 1 Gbps Ethernet LAN connection. We virtually implement two CCN routers to make connections between end-nodes (see figure 6.1). The software is implemented on Ubuntu 12.04 LTS operating system. We use Wireshark v1.8 [28] with CCN plugin installed to dissect packets in the network.

We have also designed and implemented applications to run tests on designed test bed. One is a distributed C++ application in which the participating nodes compute the consecutive numbers of fibonacci sequence in an iterative manner, that is, node 1 calculates a fibonacci number, sends the result to any other node which using this data calculates the next number in series. The application randomly selects the node to which the calculated data is to be sent and uses the proposed sender-driven technique to push the data to that node. We also develop a TCP/IP counterpart of this application to study the trade-off between CCN and TCP push-based schemes.

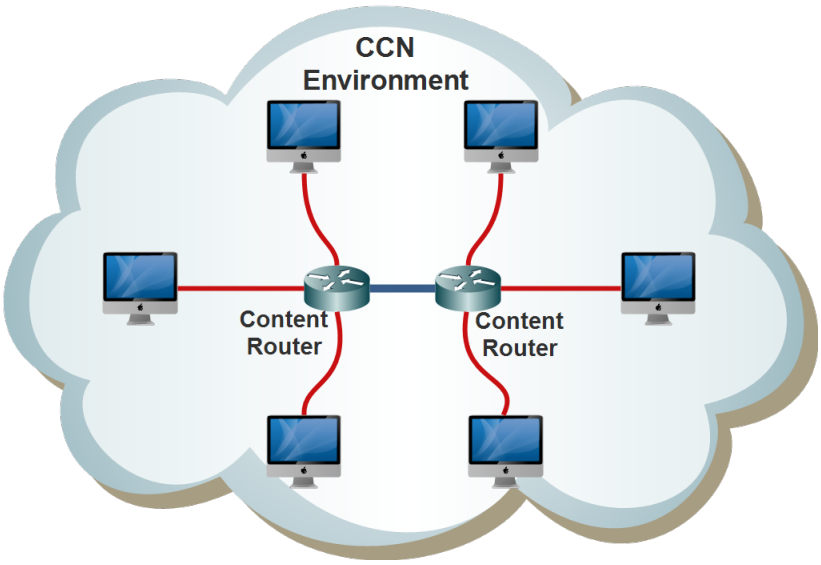

Fig. 6.1 Overview of System Structure

We use CCNAC plugin to checkpoint the above-mentioned application in CCN. However, to showcase the capability of CCNAC's checkpointing of existing CCN applications, we use a CCN enabled VLC player which can stream videos on a Content Centric Network.

6.2 Results

6.2.1 Evaluating Sender-driven Architecture

We evaluate the performance of our proposed sender-receiver application with conventional CCN client server application. The overhead of our technique has been shown in figure 6.2.

It can be seen from the result that a sender-driven communication technique accounts for ~28% of overhead when compared to receiver-driven architecture of CCN. An interesting point to note is that even though Request-To-Send is an Interest type packet, yet it takes ~1.5 times the time taken to reply to general Interest packet. On a careful study, we found out that the extra time is due to the local computation at the node to recognize a RTS Interest packet and form an Interest packet as a reply. We feel that such minimal overhead is justified for a three-way communication and will not affect the performance the system when used for extended periods.

As we were unable to find an implementation of sender-driven distributed applications using new packet types (as discussed in [14]) on CCNx, we could not compare the performance between CCNAC and new packet type applications.

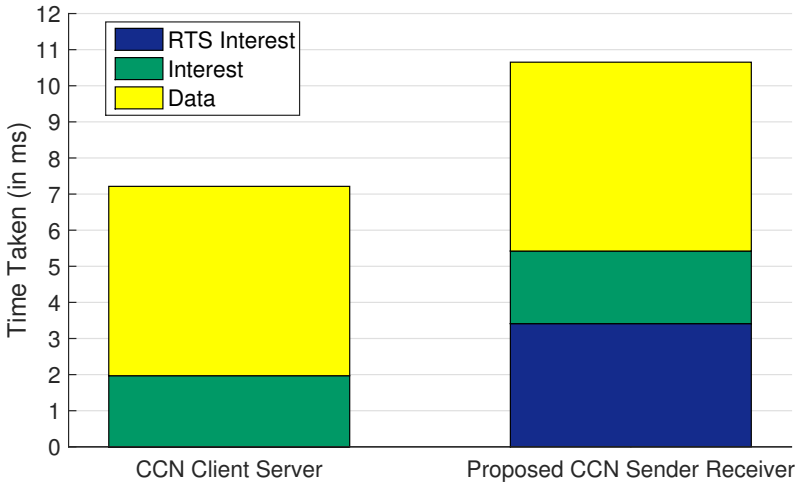

Fig. 6.2 Comparison of time taken in CCN push-based and TCP application

6.2.2 CCNAC vs. DMTCP

We now evaluate the performance of CCNAC plugin in CCN environment with standard DMTCP software in a TCP environment. We use several evaluation metrics to compare them. These have been discusses in details below:

Checkpoint Stages

We evaluate the overhead incurred on three major DMTCP stages, namely connecting end nodes to central coordinator, checkpoint request by the coordinator to end nodes and restart of end nodes from a checkpoint image. We checkpoint TCP counting application to evaluate DMTCP and CCN Fibonacci application to evaluate CCNCheck's performance. All the six nodes in the network are part of the checkpoint process for the experimentation. It is worth noting that we are only evaluating network communication time in all these stages and this value does not include memory read/write time while checkpointing. The results are depicted in figure 6.3.

It can be seen from figure 6.3 that CCNAC performs fairly equally when connecting to coordinator in a CCN environment but introduces a lot of overhead in checkpointing and checkpoint restart stages. Table 6.1 shows the increase in multiples of time in various stages between CCNAC and DMTCP.

The coordinator connect phase performs almost equally in both CCN and TCP environment and thus does not introduce much delay in the system. However, the checkpoint stage in CCN takes ~7 times the time in TCP environment. The elevation in time in checkpoint stage can be accounted to the computation time of consumer-side and producer-side algorithm

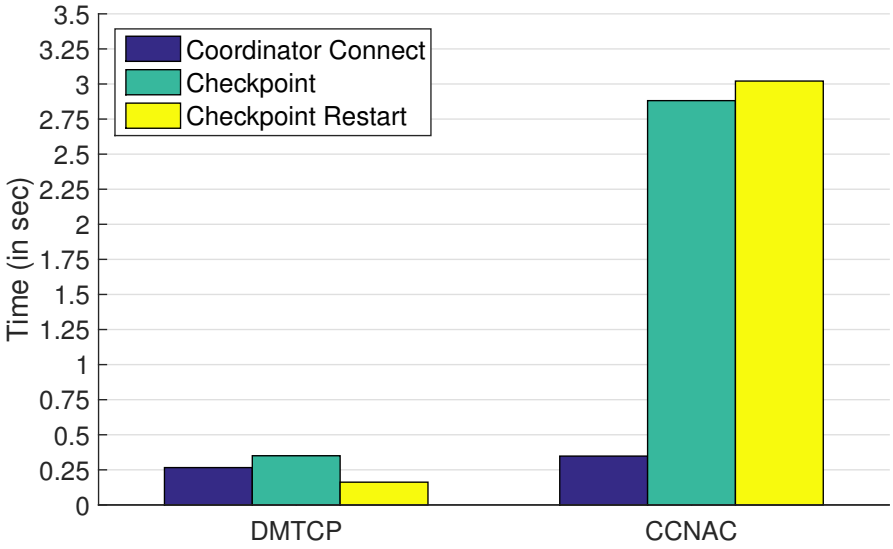

Fig. 6.3 Comparison of time taken at various checkpoint stages

Time	Increase in Time
Coordinator Connect	1.308
Checkpoint	7.24
Checkpoint Restart	18.66

Table 6.1 Increase in Time at various stages (in multiples)

which is due to the implementation of these algorithms in CCNAC. Currently, the execution of these algorithms lead to a lot of memory cycles, thus, leading to increased execution time. Flushing of CCN interests by sending flush interest packets at the time of checkpoint also plays a significant part in increased checkpoint time.

The checkpoint restart stage in CCNAC incurs the maximum overhead when compared to its TCP counterpart (~18 times). On a thorough analysis, we found that this overhead is accounted due to recreating the *"consumer table"* and *"producer table"* at restart (the computation takes ~2 seconds).

Checkpoint Size

We checkpoint fibonacci and VLC applications in TCP and VLC environments and compare the size of checkpoint image for both the applications. The results for the same has been depicted in figure 6.4.

The checkpoint image size in CCNAC exceeds that taken by DMTCP. In case of fibonacci application, we see an increase of 8.05% and in VLC media streaming, we see an increase of

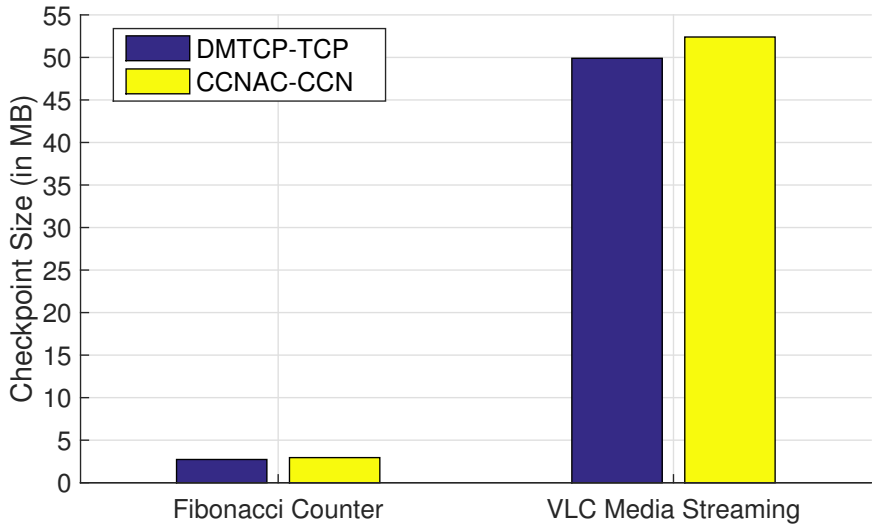

Fig. 6.4 Comparison of checkpoint image size of CCN-based and TCP-based applications

5.01%. The checkpoint image in CCNAC stores the CCN URI name of all the computing nodes with the image. We feel that this database accounts for the increased size. We feel that this increased size overhead is quite nominal and thus does not induce much system overhead.

Impact of flush packets on checkpoint time

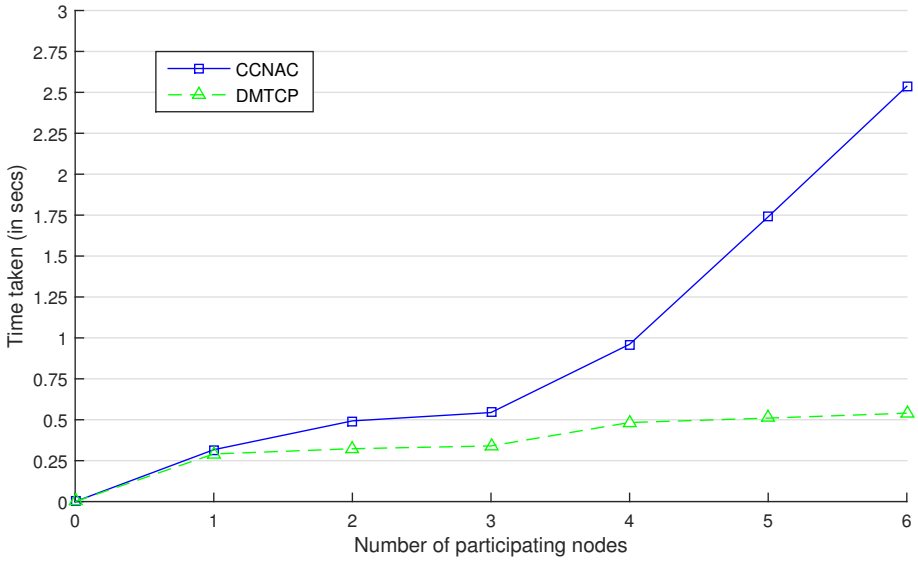

Fig. 6.5 Variation of checkpointing stage with increasing number of participating nodes

To evaluate the correlation between checkpoint time and the number of flush packets sent during checkpoint, we vary the number of nodes involved in the checkpointing process. The more the number of nodes $\propto$ flush interests sent at checkpoint. We then plot the results on a graph which has been shown in figure 6.5.

It can be observed from the graph that the checkpoint time increases somewhat linearly for nodes > 4. For nodes ≤ 4, CCNAC checkpoint stage time is somewhat equivalent to DMTCP checkpoint time. It can also be seen from the plot that unlike CCNAC, DMTCP takes almost constant time even on increasing the number of participating nodes. This result provides an evidence and justification of multitudes of increased time accounted during checkpointing stage in CCNAC.

Chapter 7

Discussions and Future Work

7.1 Discussions

7.1.1 Checkpointing using CCNCheck

Increased Checkpoint Time

In figure 6.5 of chapter 6, we observe that the time for taking a consistent checkpoint linearly increases for nodes ≥ 4. This imposes several restrictions for real-world deployment of CCNCheck.

1. **Frequent checkpointing may be unfavorable**

   The checkpointing process in CCN has an additional overhead when compared to that in TCP networks. Moreover, as checkpointing halts the execution of the application until it is complete, frequent checkpointing may not let the application progress.

2. **Checkpointing large clusters may lead to adverse results**

   As the time taken to checkpoint in linearly dependent on the size of the cluster, checkpointing distributed CCN applications on very large clusters may lead to suspension of the application.

Increased Restart Time

The checkpoint-restart for CCNCheck is ~18 times that of DMTCP. This increase can be accounted due to the computation time and memory overhead of re-initializing the consumer and producer tables at each node. Even though such a cost may seem like a huge overhead, we feel that this increased time would not affect the performance of the system. This is

because the checkpoint-restart for an application would occur only after the application has undergone a fault and needs to be restarted. We feel that such a scenario would not be a frequent occurrence for a general distributed application.

7.1.2 Sender-driven communication in CCN

The proposed sender-driven communication in CCN's imposes only 28% overhead to conventional receiver-driven communication but reduces the number of packets in the network to send the data[1] when compared to previous research. We feel that this overhead is nominal enough to deploy CCN's in a push-based environment. Also, the C++ library functions is similar to ones used by TCP/IP network developers which removes the learning curve required to understand routing of packets in CCN.

7.2 Limitations and Future Work

The work presented in this report has several limitations. Possible future works are necessary to overcome these.

1. **High Checkpoint time**: CCNCheck checkpoint time is about 7 times that of traditional DMTCP as seen in table 6.1. This is because the current implementation of CCN checkpoint algorithm is not optimized and wastes several CPU cycles on execution. One major future work would involve an efficient implementation of these algorithms.

2. **Scalability**: Due to resource constraints, we were only able to deploy CCNCheck and sender-initiated application to a testbed of six different nodes which we feel might be limited to comment on scalability of the approach. We feel that CCNCheck may perform differently (better/worse) when deployed in a distributed cluster of large size.

3. **Test applications**: We tested the performance of CCNCheck through a simple sender-initiated fibonacci counter application and a CCN VLC media stream. One can possibly do a stress testing of CCNCheck by checkpointing a large-scaled [2] distributed application such as a CCN-based distributed game as proposed by Qu et al. [30] and Wang et al. [35].

[1] from $O(n^2)$ to O(n)

[2] consisting of tens to hundreds of threads per instance

7.3 Conclusion

Applications in content centric networks follows a receiver-driven communication where a data is sent to a node only when requested. However, this poses a problem for developers to port their existing applications from TCP/IP networks to CCN as TCP-based applications follow a sender-initiated approach. We present an efficient, application-layer based sender-driven communication in CCN. Our design does not change the existing CCN architecture such as adding new packet types. We implemented our design in CCNx v0.8.2 and have exported it as C++ libraries for developers to use while porting their applications. We have also presented CCNCheck, a plugin for open-source checkpointing software DMTCP, which enables checkpointing of applications in CCN. CCNCheck works on consumer and producer side algorithms which ensures that the resulting checkpoints are always consistent in nature.

We implemented a sample application working on our proposed sender-initiated approach and compare it to existing CCN client server application. We find that our technique imposes only 28% overhead to existing receiver-driven approach which we feel is nominal. For evaluating performance of CCNCheck, we checkpointed our designed application and existing CCN VLC media player stream on the testbed. We found a minor increase of ~7% in checkpoint size when compared to its TCP counterpart. However, CCNCheck takes a considerable more time to take a checkpoint and restart from it when compared to DMTCP. We feel that this increased time is due to the implementation of designed CCN checkpoint algorithms. However, we claim that ours is the first working implementation of checkpointing distributed applications in CCN and subsequent revisions of CCNCheck will focus on efficient implementation of algorithms to reduce checkpoint time.

References

[1] CCN VLC Player Plugin. URL https://github.com/ProjectCCNx/ccnx/tree/master/apps/vlc.

[2] Failure recovery techniques in distributed systems. online lecture. URL http://www.cs.fsu.edu/~xyuan/cop5611/lecture13.html.

[3] Mobile Traffic to Hit 18 Exabytes Per Month by 2018. Online article, February 2014. URL http://www.datamation.com/networks/mobile-traffic-to-hit-18-exabytes-per-month-by-2018.html.

[4] CCNx Technical Documentation, March 2015. URL http://www.ietf.org/id/draft-mosko-icnrg-ccnxsemantics-01.txt.

[5] Bengt Ahlgren, Christian Dannewitz, Claudio Imbrenda, Dirk Kutscher, and Börje Ohlman. A survey of information-centric networking. *Communications Magazine, IEEE*, 50(7):26–36, 2012.

[6] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[7] Kapil Arya. *User-Space Process Virtualization in the Context of Checkpoint-Restart and Virtual Machines*. PhD thesis, Northeastern University Boston, 2014.

[8] Bharat Bhargava and S-R Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach. In *Reliable Distributed Systems, 1988. Proceedings., Seventh Symposium on*, pages 3–12. IEEE, 1988.

[9] Nischay Bodapati, Kaiwen Zhang, and Hans-Arno Jacobsen. Psoccn: publish/subscribe support in content-centric networking. In *Proceedings of the Posters & Demos Session*, pages 15–16. ACM, 2014.

[10] Guohong Cao and Mukesh Singhal. On coordinated checkpointing in distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 9(12):1213–1225, 1998.

[11] Guohong Cao and Mukesh Singhal. Mutable checkpoints: a new checkpointing approach for mobile computing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 12(2):157–172, 2001.

[12] Antonio Carzaniga, Michele Papalini, and Alexander L Wolf. Content-based publish/subscribe networking and information-centric networking. In *Proceedings of the ACM SIGCOMM workshop on Information-centric networking*, pages 56–61. ACM, 2011.

[13] K Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[14] Jiachen Chen, Mayutan Arumaithurai, Lei Jiao, Xiaoming Fu, and KK Ramakrishnan. Copss: An efficient content oriented publish/subscribe system. In *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*, pages 99–110. IEEE, 2011.

[15] Jiachen Chen, Lei Jiao, Mayutan Arumaithurai, Xiaoming Fu, and KK Ramakrishnan. Ps-ccn: Achieving an efficient publish/subscribe capability for content-centric networks. Technical report, Technical Report No. IFI-TB-2011-04, Institute of Computer Science, University of Goettingen, 2011.

[16] Elmootazbellah Nabil Elnozahy, David B Johnson, and Willy Zwaenepoel. The performance of consistent checkpointing. In *Reliable Distributed Systems, 1992. Proceedings., 11th Symposium on*, pages 39–47. IEEE, 1992.

[17] Ali Ghodsi, Scott Shenker, Teemu Koponen, Ankit Singla, Barath Raghavan, and James Wilcox. Information-centric networking: seeing the forest for the trees. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, page 1. ACM, 2011.

[18] Peter Gusev. NDN RTC - Audio Visual Conferencing Tool for NDN.

[19] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

[20] Van Jacobson, Diana K Smetters, Nicholas H Briggs, Michael F Plass, Paul Stewart, James D Thornton, and Rebecca L Braynard. Voccn: voice-over content-centric networks. In *Proceedings of the 2009 workshop on Re-architecting the internet*, pages 1–6. ACM, 2009.

[21] Van Jacobson, Diana K Smetters, James D Thornton, Michael F Plass, Nicholas H Briggs, and Rebecca L Braynard. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 1–12. ACM, 2009.

[22] Astrid Kiehn, Pranav Raj, and Pushpendra Singh. A causal checkpointing algorithm for mobile computing environments. In *Distributed Computing and Networking*, pages 134–148. Springer, 2014.

[23] Byoung-Jip Kim. Comparison of the existing checkpoint systems. Technical report, Technical report, IBM Watson, 2005.

[24] Junguk L Kim and Taesoon Park. An efficient protocol for checkpointing recovery in distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 4(8): 955–960, 1993.

[25] George Kola, Tevfik Kosar, and Miron Livny. Faults in large distributed systems and what we can do about them. In *Euro-Par 2005 Parallel Processing*, pages 442–453. Springer, 2005.

[26] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *Software Engineering, IEEE Transactions on*, (1):23–31, 1987.

[27] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. A data-oriented (and beyond) network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 181–192. ACM, 2007.

[28] Angela Orebaugh, Gilbert Ramirez, and Jay Beale. *Wireshark & Ethereal network protocol analyzer toolkit*. Syngress, 2006.

[29] James S Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical report, Technical Report UT-CS-97-372, Department of Computer Science, University of Tennessee, 1997.

[30] Zening Qu et al. Egal car: a peer-to-peer car racing game synchronized over named data networking. Technical report, NDN, Technical Report NDN-0010, 2012.

[31] Eric Roman. A survey of checkpoint/restart implementations. In *Lawrence Berkeley National Laboratory, Tech*. Citeseer, 2002.

[32] Joseph F Ruscio, Michael A Heffner, and Srinidhi Varadarajan. Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, 2007.

[33] Laura S Sabel and Keith Marzullo. Simulating fail-stop in asynchronous distributed systems. In *Reliable Distributed Systems, 1994. Proceedings., 13th Symposium on*, pages 138–147. IEEE, 1994.

[34] Don Towsley, Jim Kurose, and Sridhar Pingali. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. *Selected Areas in Communications, IEEE Journal on*, 15(3):398–406, 1997.

[35] Zhehao Wang, Zening Qu, and Jeff Burke. Demo overview-matryoshka: design of ndn multiplayer online game. In *Proceedings of the 1st international conference on Information-centric networking*, pages 209–210. ACM, 2014.

[36] Lixia Zhang, Deborah Estrin, Jeffrey Burke, Van Jacobson, James D Thornton, Diana K Smetters, Beichuan Zhang, Gene Tsudik, Dan Massey, Christos Papadopoulos, et al. Named data networking (ndn) project. *Relatório Técnico NDN-0001, Xerox Palo Alto Research Center-PARC*, 2010.

[37] Zhenkai Zhu and Alexander Afanasyev. Let's chronosync: Decentralized dataset state synchronization in named data networking. In *ICNP*, pages 1–10, 2013.

Appendix A

CCN Application Checkpoint (CCNAC) System Walkthrough

In this appendix, we discuss the walkthrough of CCNAC checkpointing the distributed fibonacci application over sample test bed. The walkthrough is meant to prove the step-by-step procedure of checkpointing an application on CCNAC.

1. Initializing the CCN Daemon to run CCN based application and CCNx over all the terminals. The daemon is initialized by running `ccndstart` command

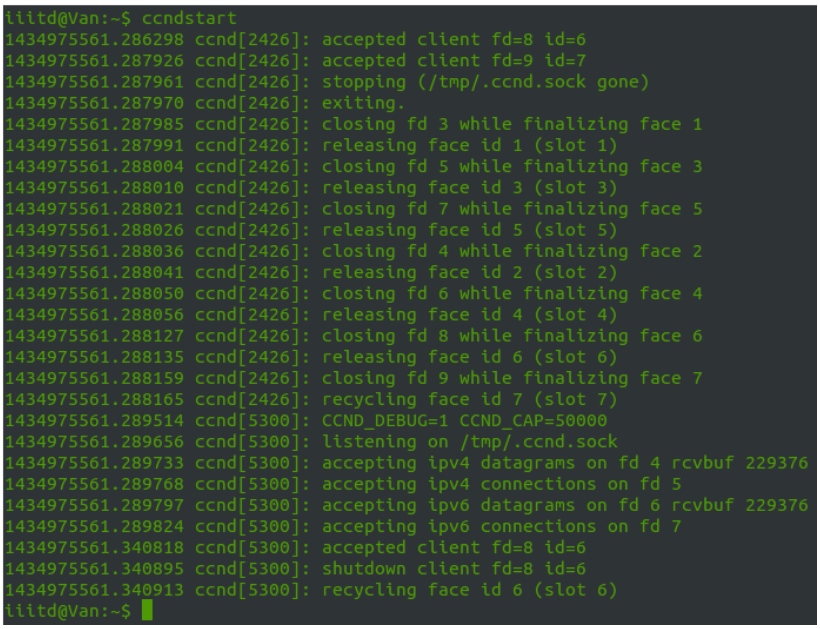

Fig. A.1 Initializing CCN Daemon

2. Initialize the CCNAC coordinator. CCNAC manages the DMTCP coordinator and empowers it to understand CCN communication channels.

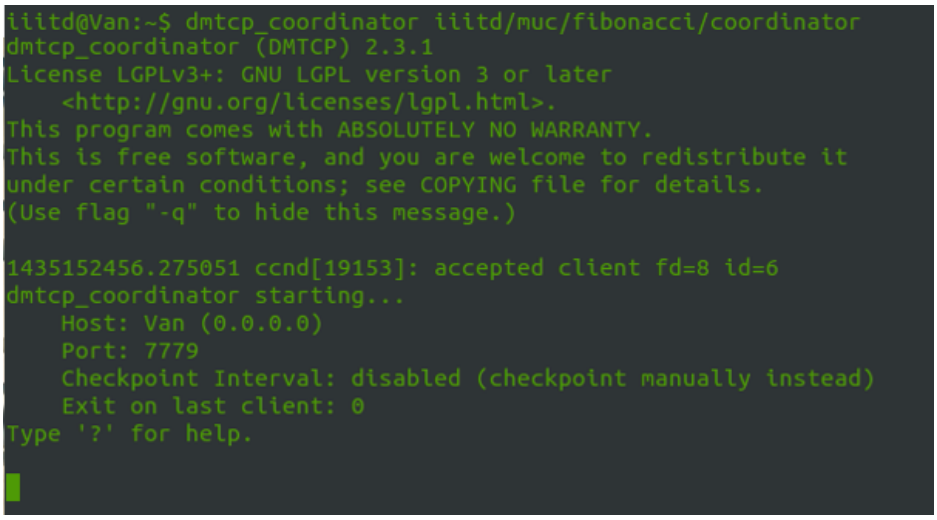

Fig. A.2 Initializing CCNAC Coordinator

3. The checkpointing of the application registered with the coordinator by explicitly giving the command `c` at the coordinator

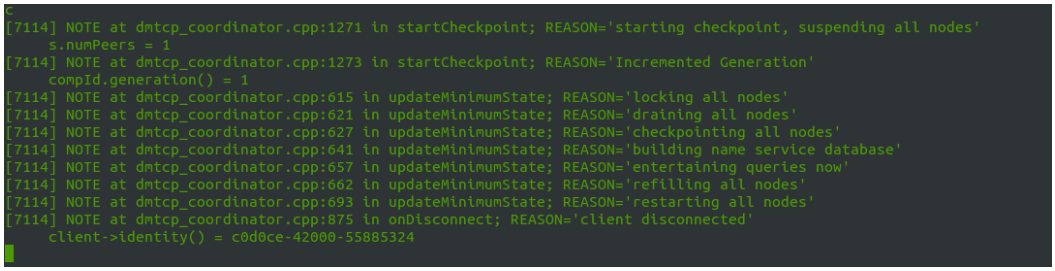

Fig. A.3 Application Checkpointing by Coordinator

4. After the checkpoint process is complete, the checkpoint image will be saved at the defined path. The image will contain the postfix of `.dmtcp`

Fig. A.4 Listing the saved checkpoint image

5. The image can be used to restart the process from the last saved computation after a transient failure.

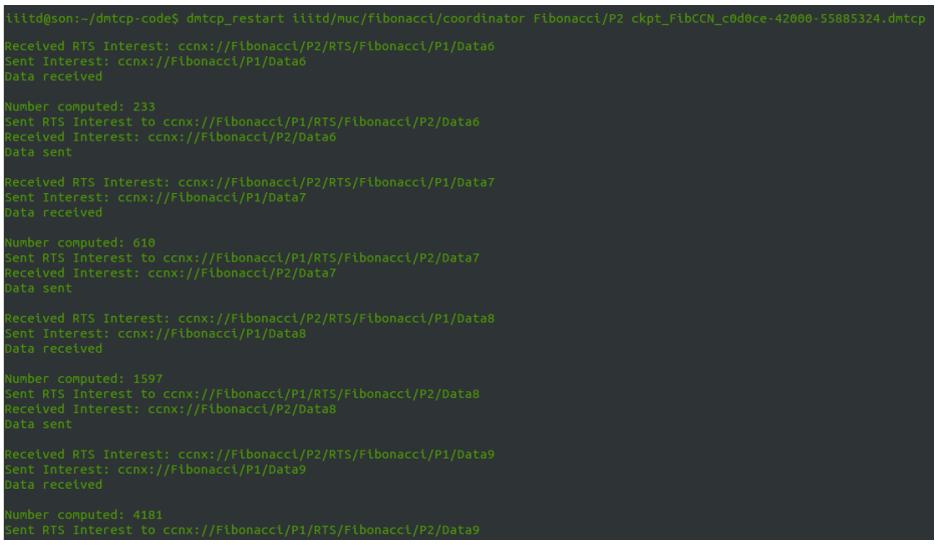

Fig. A.5 Restarting the application from the saved checkpoint

The application restarts from the saved checkpoint. As CCNAC is a plugin over DMTCP, it supports all applications which are natively supported by DMTCP.