

A TRANSACTION MODEL FOR ENVIRONMENTAL
RESOURCE DEPENDENT CYBER-PHYSICAL SYSTEMS

by

HUANG ZHU

B.S., Nanjing University of Posts and Telecommunications, China,
2008

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY

Manhattan, Kansas

2014

Abstract

Cyber-Physical Systems (CPSs) represent the next-generation systems characterized by strong coupling of computing, sensing, communication, and control technologies. They have the potential to transform our world with more intelligent and efficient systems, such as Smart Home, Intelligent Transportation System, Energy-Aware Building, Smart Power Grid, and Surgical Robot. A CPS is composed of a computational and a physical subsystem. The computational subsystem monitors, coordinates and controls operations of the physical subsystem to create desired physical effects, while the physical subsystem performs physical operations and gives feedback to the computational subsystem.

This dissertation contributes to the research of CPSs by proposing a new transaction model for Environmental Resource Dependent Cyber-Physical Systems (ERDCPSs). The physical operations of such type of CPSs rely on environmental resources, and they are commonly seen in areas such as transportation and manufacturing. For example, an autonomous car views road segments as resources to make movements and a warehouse robot views storage spaces as resources to fetch and place goods. The operating environment of such CPSs, *CPS Network*, contains multiple CPS entities that share common environmental resources and interact with each other through usages of these resources.

We model physical operations of an ERDCPS as a set of transactions of different types that achieve different goals, and each transaction consists of a sequence of actions. A transaction or an action may require environmental resources for its operations, and the usage of an environmental resource is precise in both time and space. Moreover, a successful execution of a transaction or an action requires exclusive access to certain resources.

Transactions from different CPS entities of a CPS Network constitute a schedule. Since environmental resources are shared, transactions in the schedule may have conflicts in using

these resources. A schedule must remain consistent to avoid unexpected consequences caused by resource usage conflicts between transactions. A two-phase commit algorithm is proposed to process transactions. In the pre-commit phase, a transaction is scheduled by reserving usage times of required resources, and potential conflicts are detected and resolved using different strategies, such as *Win-Lose*, *Win-Win*, and *Transaction Preemption*. Two general algorithms are presented to process transactions in the pre-commit phase for both centralized and distributed resource management environments. In the commit phase, a transaction is executed using reserved resources. An exception occurs when the real-time resource usage is different from what has been predicted. By doing internal and external check before a scheduled transaction is executed, exceptions can be detected and handled properly.

A simulation platform (CPSNET) is developed to simulate the transaction model. The simulation platform simulates a CPS Network, where different CPS entities coordinate resource usages of their transactions through a Communication Network. Depending on the resource management environment, a Resource Server may exist in the CPS Network to manage resource usages of all CPS entities. The simulation platform is highly configurable and configuration of the simulation environment, CPS entities and two-phase commit algorithm are supported. Moreover, various statistical information and operation logs are provided to monitor and evaluate the platform itself and the transaction model. Seven groups of simulation experiments are carried out to verify the simulation platform and the transaction model. Simulation results show that the platform is capable of simulating a large load of CPS entities and transactions, and entities and components perform their functions correctly with respect to the processing of transactions. The two-phase commit algorithm is evaluated, and the results show that, compared with traditional cases where no conflict resolving is applied or a conflicting transaction is directly aborted, the proposed conflict resolving strategies improve the schedule productivity by allowing more transactions to be executed and the scheduling throughput by maintaining a higher concurrency level.

A TRANSACTION MODEL FOR ENVIRONMENTAL
RESOURCE DEPENDENT CYBER-PHYSICAL SYSTEMS

by

Huang Zhu

B.S., Nanjing University of Posts and Telecommunications, China,
2008

A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY

Manhattan, Kansas

2014

Approved by:

Major Professor
Gurdip Singh

Copyright

Huang Zhu

2014

Abstract

Cyber-Physical Systems (CPSs) represent the next-generation systems characterized by strong coupling of computing, sensing, communication, and control technologies. They have the potential to transform our world with more intelligent and efficient systems, such as Smart Home, Intelligent Transportation System, Energy-Aware Building, Smart Power Grid, and Surgical Robot. A CPS is composed of a computational and a physical subsystem. The computational subsystem monitors, coordinates and controls operations of the physical subsystem to create desired physical effects, while the physical subsystem performs physical operations and gives feedback to the computational subsystem.

This dissertation contributes to the research of CPSs by proposing a new transaction model for Environmental Resource Dependent Cyber-Physical Systems (ERDCPSs). The physical operations of such type of CPSs rely on environmental resources, and they are commonly seen in areas such as transportation and manufacturing. For example, an autonomous car views road segments as resources to make movements and a warehouse robot views storage spaces as resources to fetch and place goods. The operating environment of such CPSs, *CPS Network*, contains multiple CPS entities that share common environmental resources and interact with each other through usages of these resources.

We model physical operations of an ERDCPS as a set of transactions of different types that achieve different goals, and each transaction consists of a sequence of actions. A transaction or an action may require environmental resources for its operations, and the usage of an environmental resource is precise in both time and space. Moreover, a successful execution of a transaction or an action requires exclusive access to certain resources.

Transactions from different CPS entities of a CPS Network constitute a schedule. Since environmental resources are shared, transactions in the schedule may have conflicts in using

these resources. A schedule must remain consistent to avoid unexpected consequences caused by resource usage conflicts between transactions. A two-phase commit algorithm is proposed to process transactions. In the pre-commit phase, a transaction is scheduled by reserving usage times of required resources, and potential conflicts are detected and resolved using different strategies, such as *Win-Lose*, *Win-Win*, and *Transaction Preemption*. Two general algorithms are presented to process transactions in the pre-commit phase for both centralized and distributed resource management environments. In the commit phase, a transaction is executed using reserved resources. An exception occurs when the real-time resource usage is different from what has been predicted. By doing internal and external check before a scheduled transaction is executed, exceptions can be detected and handled properly.

A simulation platform (CPSNET) is developed to simulate the transaction model. The simulation platform simulates a CPS Network, where different CPS entities coordinate resource usages of their transactions through a Communication Network. Depending on the resource management environment, a Resource Server may exist in the CPS Network to manage resource usages of all CPS entities. The simulation platform is highly configurable and configuration of the simulation environment, CPS entities and two-phase commit algorithm are supported. Moreover, various statistical information and operation logs are provided to monitor and evaluate the platform itself and the transaction model. Seven groups of simulation experiments are carried out to verify the simulation platform and the transaction model. Simulation results show that the platform is capable of simulating a large load of CPS entities and transactions, and entities and components perform their functions correctly with respect to the processing of transactions. The two-phase commit algorithm is evaluated, and the results show that, compared with traditional cases where no conflict resolving is applied or a conflicting transaction is directly aborted, the proposed conflict resolving strategies improve the schedule productivity by allowing more transactions to be executed and the scheduling throughput by maintaining a higher concurrency level.

Table of Contents

Table of Contents	viii
List of Figures	xii
List of Tables	xiv
Acknowledgements	xv
Dedication	xvi
Preface	xvii
1 Introduction	1
1.1 Cyber-Physical System	1
1.2 ERDCPS	3
1.3 Our Approach	5
1.4 Contributions	6
1.5 Organization	8
2 Problem Definition	9
2.1 Problem Definition	10
2.2 Research Objectives and Challenges	11
3 Related Work	15
3.1 Introduction	15
3.2 Event-Based Models	16
3.2.1 Overview of Event-Based Models	16
3.2.2 Discussion	17
3.3 Service-Based Models	18
3.3.1 Overview	18
3.3.2 Discussion	19
3.4 Agent-Based Models	20
3.4.1 Overview	20
3.4.2 Discussion	21
3.5 Resource Sharing	21
3.5.1 Real-Time Systems	21
3.5.2 Traffic Management Systems	23
3.6 Candidate: Transaction Model	24

3.6.1	Summary	27
4	A Transaction Model for CPS	29
4.1	Introduction	29
4.2	A Sample Transaction	29
4.3	CPS and CPS Network	30
4.4	Actions	33
4.5	Transactions	37
4.6	Schedules	39
4.7	Summary	41
5	Transaction Processing	43
5.1	Introduction	43
5.2	Two-Phase Commit Transaction Processing	44
5.2.1	Transaction pre-Write Set	44
5.2.2	Two-Phase Commit	45
5.2.3	Transaction State Machine	46
5.2.4	Action-Level Two-Phase Commit	48
5.3	Pre-Commit Phase and Conflicts	49
5.3.1	Conflict Types	49
5.3.2	Transaction Precedence	50
5.3.3	PP Conflict and Resolution Strategy	50
5.3.4	PC Conflict and Resolution Strategy	52
5.3.5	Dynamic Transaction Adjustment	53
5.3.6	Pre-Commit Phase: Conflicts in Multiple Transactions	54
5.4	Pre-Commit Phase: Transaction Processing Algorithm	54
5.4.1	State of a Reservation	55
5.4.2	Resource Client and Server in Centralized Environment	56
5.4.3	Resource Client and Server in Distributed Environment	58
5.5	Commit Phase and Exceptions	62
5.5.1	Exception	62
5.5.2	Exception Detection and Handling	63
5.5.3	Runtime Exception	65
5.6	Summary	65
6	Simulation Platform Implementation: CPSNET	67
6.1	Introduction	67
6.2	Resource and Resource Usage	71
6.2.1	Resource	71
6.2.2	Resource Usage	72
6.2.3	Resource Usage Operations	73
6.3	Action and Transaction	74
6.3.1	System State	74

6.3.2	Action	74
6.3.3	Transaction	77
6.4	Cyber-Physical System	79
6.4.1	Transaction Processing Flow	82
6.4.2	Transaction Generator	82
6.4.3	Transaction Scheduler	84
6.4.4	Transaction Executor	87
6.4.5	Resource Manager	88
6.5	Resource Server	89
6.5.1	Tasks of a Resource Server	89
6.5.2	Message	92
6.5.3	Reservation State Machine	93
6.6	Communication Network	94
6.7	CPS Network	96
6.7.1	Configuration	96
6.7.2	Time-Based Simulation	97
6.7.3	Statistical Information	98
6.8	Summary	98
7	Simulation and Results	100
7.1	Introduction	100
7.2	Assumptions and Configurations	102
7.3	Performance Evaluation Criteria	105
7.4	Simulation Cases	106
7.5	Group 0: Verifying The Simulation Platform	108
7.6	Group 1-5: Verifying The Conflict Resolving Algorithms	112
7.6.1	Group 1: ConstantSpeedTransaction	113
7.6.2	Group 2: AccelerateTransaction	115
7.6.3	Group 3: DecelerateTransaction	117
7.6.4	Group 4: ChangeToLeftLaneTransaction	119
7.6.5	Group 5: ChangeToRightLaneTransaction	120
7.6.6	Discussion	121
7.7	Group 6: A Comprehensive Simulation	122
7.7.1	Simulation Scenario	123
7.7.2	Simulation Result	124
7.7.3	Statistics	128
7.7.4	Scalability	133
7.8	Summary	138
8	Conclusions and Future Work	141
8.1	Conclusions	141
8.2	Future Work	143

Bibliography	146
A Sample Simulation Result of CyberPhysicalSystem	154
B Sample Simulation Result of CPSNetwork	163

List of Figures

1.1	Cyber-Physical System: A Concept Map	2
1.2	A CPS Network	4
3.1	Comparison of Existing Models and ERDCPSs	28
4.1	An Instance of Transaction ChangeToRightLane	30
4.2	Cyber-Physical System	31
4.3	Usage of Environmental Resources by Actions	35
5.1	Transaction State Machine	47
5.2	Weighing Process Between T_i and T_j	51
5.3	State Machine of Resource Reservation	55
5.4	Messages Between Client and Server	56
5.5	Algorithm for Resource Server: Receiver of REQ_c	59
5.6	Algorithm for Resource Client: Sender of REQ_c	61
6.1	CPSNET Simulation Platform	68
6.2	Simulation Flow	70
6.3	Resource, ResourceUsage, and ResourceUsageOperations	71
6.4	Action	75
6.5	Transaction	77
6.6	Architecture of Cyber-Physical System	79
6.7	Transaction Processing Flow	83
6.8	Transaction Generator	83
6.9	Transaction Scheduler	85
6.10	Transaction Executor	87
6.11	Resource Manager	88
6.12	Resource Server	90
6.13	Reservation and Reservation State	91
6.14	Message	93
6.15	Reservation State Machine	93
6.16	Communication Network	94
6.17	External and Internal Message Transmission	95
6.18	CPSNetwork	96
6.19	Statistical Information of CPSNetwork	99
7.1	Examples of Configuration Files	103
7.2	Simulation Group 0	109

7.3	Simulation Scenario and Result of Group 0	110
7.4	Operation Log of TransactionScheduler	111
7.5	Conflict Resolver	113
7.6	Simulation Group 1	114
7.7	Simulation Group 2	116
7.8	Simulation Group 3	118
7.9	Simulation Group 4	119
7.10	Simulation Group 5	120
7.11	Group 6 - Scenario	122
7.12	Group 6 - CPS Configuration	123
7.13	Group 6 - Simulation Result of CPS Network	125
7.14	Group 6 - Simulation Result of 10 Runs	125

List of Tables

7.1	Simulation Test Cases	106
7.2	Statistics of Simulation Group 6	130
7.3	Simulation Result of Scaling Experiments	135
7.4	Statistics of Scaling Experiments	136

Acknowledgments

The six years of time I have spent pursuing the PhD degree in Kansas State University is one of the most rewarding and interesting experience in my life. The road to the doctorate is full of challenges, but it is a happy journey for me because of the encouragement, support and love from many people.

First, I would like to express my sincere gratitude to my major professor, Gurdip Singh, for his guidance, support and trust. I am fortunate enough to work under his supervision. Dr. Singh has been a very resourceful and helpful person. I am very grateful for his willingness to share his knowledge to help me learn and master research skills, his guidance on my research work whenever I was stuck and frustrated, and the freedom he gave me to develop a solid research background and broaden my knowledge in many fields.

Second, I would like to thank all outstanding committee members, Professor Torben Amtoft, Professor Daniel Andresen and Professor Don Gruenbacher, for their advice on my research and their time attending my Research Proficiency Exam, Research Proposal Defense and Dissertation Defense.

Third, I would like to thank my colleagues and friends. Special thanks to Jing Xia for helping me adapt to the life here when I first came to Manhattan in 2008; to Zhi Zhang and Hongmin Li for treating me to all delicious feasts in their place, and I really enjoyed the time we hanged out and went fishing together; to Rui Zhuang for sharing new ideas and technologies with me, and some day we will found a startup together.

Last but not least, I would like to thank my family, especially my mother and father, for their understanding, love and support throughout my life.

Dedication

To my mother, Yiyang Zhu, and my father, Ruiliu Zhu.

Preface

This dissertation is submitted for the degree of Doctor of Philosophy at the Kansas State University. The research presented in this dissertation was conducted under the supervision of Professor Gurdip Singh in Department of Computing and Information Sciences, Kansas State University, between August 2008 and August 2014.

The work presented in this thesis is original, to the best of my knowledge, except where acknowledgments and references are made to previous work. I hereby declare that I have not submitted this material, either in full or in part, for a degree at this or any other institution.

Part of this work has been presented in following publications:

1. Zhu, Huang, and Gurdip Singh. “A communication protocol for a vehicle collision warning system.” In *Proceedings of the 2010 IEEE/ACM Int’l Conference on Green Computing and Communications & Int’l Conference on Cyber, Physical and Social Computing*, pp. 636-644. IEEE Computer Society, 2010.
2. Zhu, Huang, Sumeet Gujrati and Gurdip Singh. “A Transaction Model for Environment-Resource Dependent Cyber-Physical Systems.” In Submission.
3. Zhu, Huang, and Gurdip Singh. “A Simulation Platform for Environment-Resource Dependent Cyber-Physical Systems: CPSNET.” In Preparation.

Chapter 1

Introduction

1.1 Cyber-Physical System

The concept of Cyber-Physical Systems (CPSs) has been proposed to develop next-generation systems characterized by a strong integration of the computing, sensing, communication, and control technologies [1, 2, 10]. A CPS is composed of a computational and a physical subsystem. The computational subsystem consists of software components, and the physical subsystem is composed of various physical devices. The computational subsystem monitors, coordinates and controls operations of the physical subsystem to create desired physical effects, while the physical subsystem performs physical operations and gives feedback to enable the computational subsystem to perform computations to find solutions for emerging problems.

The tight integration of the physical and computational subsystems creates new engineering systems with more intelligence, adaptability, reliability, precision, and robustness. CPSs are destined to transform our world with more intelligent and efficient systems, such as Smart Home, Intelligent Transportation System, Energy-Aware Building, Smart Power Grid, and Surgical Robot. Beneath the promising future as we can envision of CPSs, however, the development of CPSs relies on great improvements of today's technologies in many areas. Figure 1.1 [3] states what supporting technologies are required and what challenges are confronted.

Cyber-Physical Systems - a Concept Map <http://CyberPhysicalSystems.org> See authors and contributors.

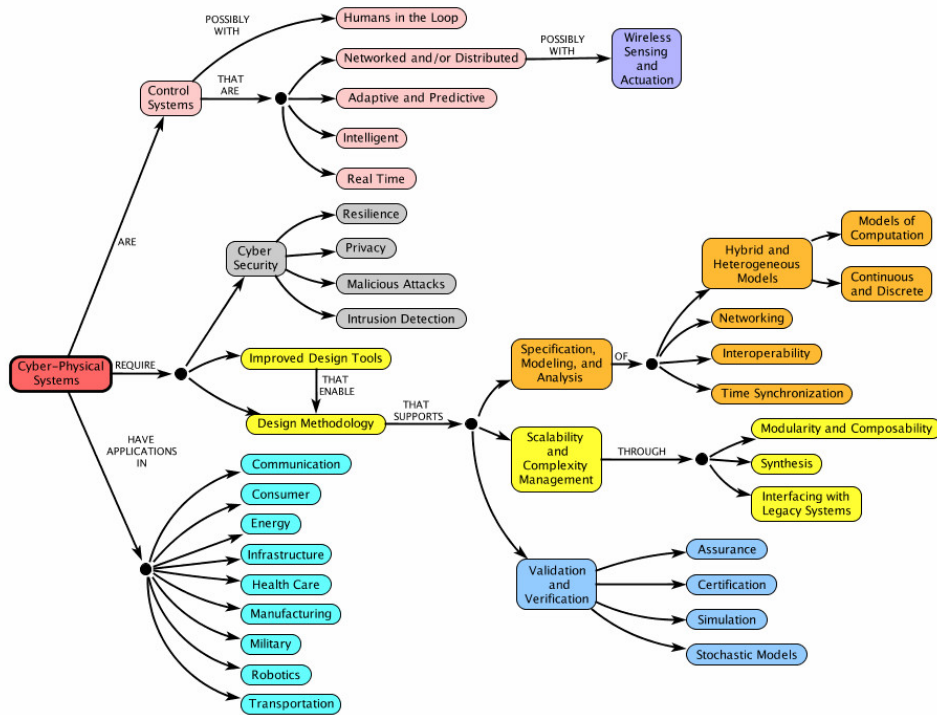


Figure 1.1: Cyber-Physical System: A Concept Map

There has been a number of recent efforts to examine current technologies for CPSs and to identify potential challenges and research needs [4–8]. Much work have also been done in specific domains to recognize impeding difficulties such as industrial control [11], medical systems [12–14], transportation [15–18], power grid [19, 20], and environment monitoring [21]. Especially, Proceedings of The IEEE held a special issue on CPSs [9] to discuss ongoing research towards CPSs in different areas such as group control and management [22, 23] and control system modeling and integration [36, 37].

1.2 ERDCPS

Our research concentrates on a particular type of CPSs, Environmental Resource Dependent Cyber-Physical System (ERDCPS), which are commonly seen in areas such as transportation and manufacturing such as autonomous cars, warehouse robots and assembling robots.. Physical operations of an ERDCPS rely on environmental resources. For example, an autonomous car views road segments as resources to make movements, a warehouse robot views storage spaces as resources to fetch and place goods, and an assembly line robot views materials or parts as resources to produce goods. Environmental resources indicates external resources that are beyond the control of any ERDCPS. The operating environment of an ERDCPS usually includes other peer ERDCPSs, and all ERDCPSs share the same environmental resources for their physical operations. These ERDCPSs together constitute a network, named *CPS Network*, and they interact with each other through usages of environment resources, as shown in Figure 1.2.

The usage of an environmental resource by a physical operation of an ERDCPS is precise in both time and space. It not only specifies which resource is needed, but also indicates the time period during which the resource is used. When a resource is being used, it can be used in a shared or exclusive manner. In this dissertation, we are only dealing with cases where resource usage is exclusive. Since environmental resources are shared in a CPS Network, if operations from more than one ERDCPS access a particular resource at the same time, a

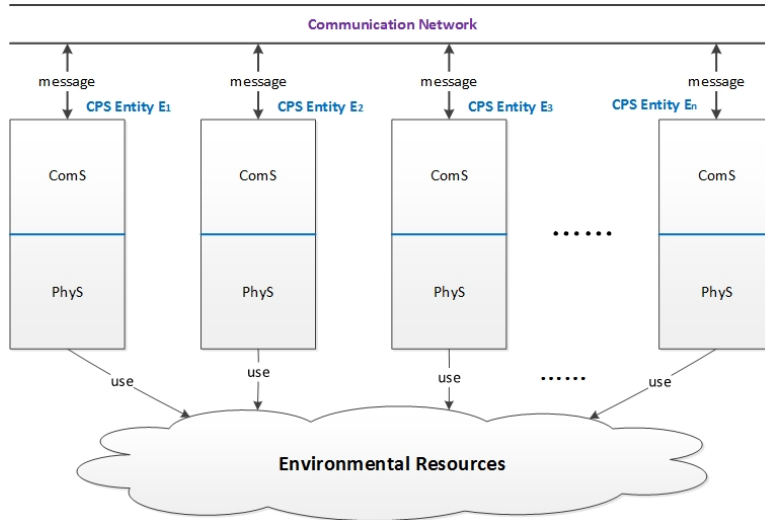


Figure 1.2: A CPS Network

conflict occurs. Conflicts can have bad consequences, e.g., a resource usage conflict between two cars can cause a collision. To prevent undesired consequence, an ERDCPS should be able to detect and resolve potential resource usage conflicts. If there exists a separate entity in the network that works as a resource manager, then each ERDCPS communicates with it to check resource usage conflicts. In the absence of such a resource manager entity, every ERDCPS then coordinates with each other to use resources without causing conflicts. Once potential conflicts are detected, they should be resolved.

Existing models for CPSs, such as event-based, service-based and event-based models, have limitations in capturing the characteristics of ERDCPSs. For example, the environmental-resource-dependent feature of a physical operation of an ERDCPS is absent in all existing models we have examined, and the operating environments of CPSs defined in existing models do not contain other peer CPSs that compete for usages of environmental resources. Given the deficiency of existing models, we propose a new transaction model for ERDCPSs. While the concept of transaction is mostly used in systems such as banking and databases to indicate a all-or-none atomic operation, our definition of transaction here is more structure-oriented. We consider a transaction as a collection unit of physical operations that achieve a certain goal and the atomicity constraint is not necessarily followed.

1.3 Our Approach

In this dissertation, we present a transaction model for ERDCPSs. The operating environment of such CPSs, *CPS Network*, contains multiple CPS entities that share common environmental resources and interact with each other through usages of these resources. We model physical operations of an ERDCPS as a set of transactions of different types that perform different tasks. Each transaction consists of a sequence of actions of different types with each performing a sub-task. A transaction or an action may require environmental resources for its physical operations, and a successful execution of a transaction or an action requires exclusive access to certain resources when they being used. Transactions from different CPS entities of a CPS Network constitute a schedule. A schedule must remain consistent to avoid unexpected consequences caused by resource usage conflicts between transactions.

We propose a two-phase commit algorithm to process transactions from each CPS entity. In the pre-commit phase, a transaction is scheduled by reserving its pre-Write set, which indicates the predicted usage times of required resources. A potential conflict occurs when the pre-Write sets of two transactions overlap. Depending on the state of a transaction, different conflict types are defined and several strategies are introduced to resolve them, such as *Win-Lose*, *Win-Win*, and *Transaction Preemption*. Two general algorithms are presented to process transactions in the pre-commit phase in the centralized and distributed resource management environments respectively. In the commit phase, a transaction is executed using reserved resources. An exception occurs when the real-time resource usage is different from what has been predicted. By doing internal and external check before a scheduled transaction is executed, internal and external exceptions can be detected and handled properly.

To simulate the transaction model, a simulation platform (CPSNET) is developed. The simulation platform simulates a CPS Network, where different CPS entities coordinate resource usages of their transactions through a Communication Network. Depending on the

resource management environment, a Resource Server may exist in the CPS Network to manage resource usages of all CPS entities. The simulation platform is highly configurable and configuration of the simulation environment, CPS entities and two-phase commit algorithm are supported. Various statistical information and operation logs are provided to monitor and evaluate the simulation platform itself and the transaction model. algorithm performance. Seven groups of simulation experiments are carried out to verify the simulation platform and the transaction model using CPSNET. Simulation results show that the platform is able to simulate a large load of CPS entities and transactions, and each entity and component perform their functions correctly with respect to the processing of transactions. Moreover, the two-phase commit algorithm is evaluated, and the results show that, compared with traditional cases where no conflict resolving is applied or direct abortion of a conflicting transaction is used, the proposed conflict resolving strategies improve schedule productivity by allowing more transactions to be executed and schedule throughput by maintaining a higher transaction concurrency level.

1.4 Contributions

The contributions of this dissertation include:

1. We explicitly define ERDCPS, a particular application type of CPSs, whose physical operations rely on environment resources to create desirable physical effects. We also define its operating environment, CPS Network, which contains multiple ERDCPSs that share the same environment resources and interact with each other through usages of these resources.
2. We propose a transaction model to model the physical operations of an ERDCPS. Two levels of abstractions are defined to model the physical operations: action and transaction. A transaction is composed of a sequence of actions. We also define the resource-dependent feature of an action and a transaction.

3. We design a two-phase commit algorithm to process transactions in order to detect and resolve potential conflicts. In the pre-commit phase, a transaction is scheduled and potential transaction conflicts are detected and resolved. In the commit phase, potential exceptions are detected and handled before a transaction is executed.
4. We propose several conflict resolving and exception handling strategies to avoid transaction conflicts. Conflict resolving is applied in the pre-commit phase, and strategies include Win-Lose, Win-Win, Enhanced Win-Win, and Transaction Preemption. Exception handling strategies are used in the commit phase, and they include internal and external detection and handling of potential exceptions.
5. We present two general algorithms that work in the centralized and distributed resource management environments respectively. In the centralized environment, a Resource Server exists and all CPS entities communicate with the server to reserve required resources. In the distributed environment, CPS entities coordinate their resource usages without a central Resource Server. Each CPS entity is both a resource client and server. When it reserves resources for its transactions, it is a resource client. When it handles reservation requests from other CPS entities, it is a resource server.
6. We implement a highly configurable simulation platform for the transaction model: CPSNET. The platform simulates a CPS Network consisting of CPS entities, a Resource Server, and a Communication Network. In a simulation, each entity operates independently and communicates with others using messages. In each CPS entity, the two-phase commit algorithm is used to process transactions. Two levels of configuration (simulation environment and CPS) are supported by CPSNET, which allow simulations under different settings such as different conflict resolving strategies. Various statistical information and operation logs are provided and they make the analysis of simulation results much convenient. The modularized design of CPSNET makes it easy to be ported to other applications.

7. Several groups of simulations are performed using the CPSNET platform. Experiment results show that the simulation platform carries out a simulation successfully, each entity does its job as expected, and all components of a CPS entity fulfill their functions correctly with respect to the two-phase commit transaction processing algorithm. The results also show that the proposed conflict resolving strategies improve schedule throughput and productivity.

1.5 Organization

The rest of the dissertation is organized as follows:

- Chapter 2 defines the problem of our research, and identifies our research objectives and challenges.
- Chapter 3 gives a brief overview of related work and discusses why existing models are not directly used in our research.
- Chapter 4 proposes a new transaction model for an ERDCPS and explicitly defines the concepts of CPS, CPS Network, Action, Transaction, Schedule, etc.
- Chapter 5 presents a two-phase commit transaction processing algorithm. It shows how a transaction is scheduled and executed, and how potential transaction conflicts and exceptions are detected and resolved. Besides, two general algorithms working in the centralized and distributed resource management environment are proposed.
- Chapter 6 introduces how the CPSNET simulation platform is implemented.
- Chapter 7 discusses our simulation experiments and analyzes the simulation results with respect to the simulation platform and the transaction model.
- Chapter 8 concludes this dissertation and discusses the future work.

Chapter 2

Problem Definition

This dissertation focuses on Environmental Resource Dependent Cyber-Physical System (ERDCPS). This type of CPSs are commonly seen in areas such as transportation and manufacturing, where physical operations rely on environmental resources. For example, an autonomous car views road segments as resources to make movements, a warehouse robot views storage spaces as resources to fetch and place goods, and an assembly line robot views materials or parts as resources to produce goods.

An application example of ERDCPS is an autonomous car. An autonomous car operates on its own and has the capabilities to navigate itself in a group of autonomous cars on a road. It is composed of two subsystems: physical and computational. The physical subsystem is composed of communication devices (e.g., DSRC radio [56]), sensors that read speed and acceleration, detect positions and calculate inter-vehicle distance, actuators that turn on/off signals and trigger operation of mechanical systems, and a mechanical system that carries out physical operations such as turning and moving. The main task of the physical subsystem is to perform physical operations and to create the desired physical effects, i.e., movements on a road. The computational subsystem controls operations of the physical subsystem. It calculates proper speed or acceleration for the physical system to apply, gives corresponding instructions to the physical subsystem to execute, and monitors operations carried out by the physical subsystem. Physical operations (or movements) of a car includes operations such as turning right or left, increasing or decreasing speed, and moving forward or backward. These

physical operations all rely on road segments, the environmental resources, to complete their movements. The operating environment of a car contains other cars that share the same road to make movements. If two or more cars use the same road segment at the same time, a resource usage conflict occurs and a car collision will be caused. Thus, the execution of a physical operation of a car requires exclusive access to the road segment when it is being used. To guarantee the safety of each car, all cars moving on the road must coordinate with each other to avoid resource usage conflicts.

2.1 Problem Definition

Because of the specialization of an ERDCPS, its physical operations usually follow certain patterns. For example, a car's physical operations are limited to certain moving patterns, such as turning, accelerating, decelerating, and moving at a constant speed. Moreover, physical operations are constrained by the capabilities of the physical subsystem, e.g., a car's speed cannot exceed the maximum speed that the mechanical system can achieve. They also have to follow rules of the operating environment, e.g., a car is supposed to move on a road, and cannot change to its right lane if it is already on the right-most lane of a road.

Physical operations performed by an ERDCPS form a process which has a specific goal to achieve, such as a car moving from point A to point B on a road R. The process can be further divided into a sequence of sub-processes, each achieving a sub-goal. For example, a car's moving process from A to B is composed of sub-processes such as accelerating, decelerating, constant speed moving, changing lanes, and making turns. Each sub-process can be further divided into smaller units, e.g., making a right turn is composed of the following operations: turning on indicator, turning wheels clockwise to make the turn, turning wheels anti-clockwise to readjust direction, and finally turning off indicator.

The operating environment of an ERDCPS usually includes other peer ERDCPSs, and they share the same environmental resources. These ERDCPSs together constitute a net-

work and interact with each other through usages of environment resources (Figure 1.2), e.g., a group of autonomous cars using the same road for their movements, a group of warehouse robots the same storage spaces to store goods, and manufacturing robots of an assembly line assemble products using the same set of materials or parts.

The usage of an environmental resource by a physical operation of an ERDCPS is sensitive in both time and space. It specifies not only which resource is needed, but also which time period the resource is used. Moreover, when a resource is being used, exclusive access is required. Since all ERDCPSs in a network share environmental resources, if operations from more than one ERDCPS access a particular resource at the same time, a conflict occurs. Conflicts always cause bad consequences, e.g., a resource usage conflict between two cars indicates a collision. To prevent undesired consequence, an ERDCPS should be able to detect and resolve potential resource usage conflicts. This requires physical operations of an ERDCPS to be checked for conflicts before it is executed.

In a CPS Network, there may exist a central Resource Server who manages resource usages of all CPSs in the network. If such a server exists, then a CPS coordinates resource usages of its transactions with the server, and the server is responsible for detecting potential resource usage conflicts and telling the corresponding CPS whether its transaction will cause conflicts. If no such server exists, every CPS then has to coordinate its resource usages with all other CPSs in the network. In this case, a CPS is both a resource client and server. When it reserves resources for its transactions, it is a resource client. When it handles reservation requests from other CPS entities, it is a resource server. When a central server exists, we call the operation environment as a **centralized** resource management environment. Otherwise, it is a **distributed** resource management environment.

2.2 Research Objectives and Challenges

Our research objective is to build a model for ERDCPSs which captures characteristics of an ERDCPS and its operating environment discussed in the previous section. It includes

the following tasks.

1. Model the operating environment of an ERDCPS. The operating environment involves multiple ERDCPSs that share the same environmental resources. The challenges are:
 - Define roles of different entities in the environment, e.g., ERDCPSs, environmental resources, and possible resource servers that manage environmental resources.
 - Define interactions between entities in the operating environment, e.g., how resources are managed by resource servers and used by an ERDCPS, how the usages of resources by an ERDCPS affect others.
 - The representation of entities, especially environmental resources. Since all ERDCPSs in the operating environment share the same resources, a common representation of these resources is required.
2. Build an architecture for an ERDCPS. The architecture includes the computational and the physical subsystems. Not only should their compositions be investigated, but also interactions between them. Questions such as how operations of both the computational and the physical subsystems are defined, how the computational subsystem views the physical subsystem and vice versa, and how the control-feedback loop work, have to be answered.
3. Model operations of an ERDCPS. Operations of an ERDCPS are goal-driven, following certain patterns, and environmental-resource-dependent. The following questions have to be answered by our model:
 - The connection between goals and operations. How a goal is represented and transformed into a sequence of operations that accomplish the goal?
 - How an operation is defined and what properties does an operation have?

- What patterns or rules does an operation have? What effects do they have on an operation? How patterns or rules of an operation are embedded into its definition?
 - The relation between an operation and environmental resources. How resources are used by an operation? What requirements on resources does an operation have?
 - What is the operation context for an operation? How it affects an operation and further the physical effects that an operation creates?
4. Design operation processing algorithms. When a goal is selected for an ERDCPS, corresponding operations are scheduled to be executed to fulfil the goal. The processing of operations involves the computational and the physical subsystem. Questions such as how an operation is scheduled and executed and what roles the computational and the physical subsystems play in this process need to be answered.
5. Develop coordination protocols for different ERDCPSs within the operating environment. Since different ERDCPSs share common resources, protocols are needed to coordinate their operations and resource usages. Coordination protocols should consider the following aspects:
- How resource usage conflicts are detected and resolved.
 - What coordination is required for an ERDCPS both internally and externally. Internal coordination indicates detecting and resolving conflicts when scheduling or executing each operation in an ERDCPS, while external coordination indicates cooperation between different ERDCPSs to avoid and resolve conflicts.
 - What actions to take when conflicts occur in order to reduce the impact of consequences.

6. Evaluate and verify the model and protocols through simulations. The following aspects should be considered:

- Which simulation platform to used. Does it meet the requirements of our models? The selection should match characteristics of an ERDCPS and its operating environment.
- What properties of a model or a protocol to be examined and what performance is desired should be specified.
- Evaluation criteria must be developed to measure performance.
- Given the simulation results, what can be done to improve the model.

Chapter 3

Related Work

3.1 Introduction

Research on modeling CPS has primarily focused in three directions: architecture, operation, and engineering. An architectural model defines how different components of a CPS are composed or integrated. These components, Cyber and Physical, may be heterogeneous and sometimes cross-domain. An operational model defines operation patterns of a CPS, including both internal and external operations. Internal operations indicate interactions between different components of a CPS, while external operations indicate interactions between a CPS and its operating environment. Engineering models focus on the control and dynamics of a CPS. They study the stochastic nature of communication systems, discrete dynamics of computing systems, and continuous dynamics of control systems [10].

Our research focus on modeling the architecture and operation of CPSs. In this chapter, we examine existing architectural and operational CPS models and related works.

We first give an overview of existing event-based, service-based and agent-based CPS models, and discuss the reason that we don't select them to model ERDCPSs. Then we investigate related works in resource sharing, and discuss their difference from environmental resource sharing in ERDCPSs. After that we describe why the transaction-based model is selected for ERDCPSs, and describe its difference from database transaction models.

3.2 Event-Based Models

Event-based models [24–28] treat a CPS as an event-driven system, where events act as triggers. An event is a condition of interest. When an event is detected, corresponding handling operations are executed to react to the situation indicated by the perceived event. Event-based models capture interactions between a CPS and its operating environment.

3.2.1 Overview of Event-Based Models

The notion of event can be defined for different purposes [24]. For example, an event may indicate sending or receiving a message in an actor model of computation, an action shared by two processes in a process algebra model of computation, or an occurrence in time and space in the world of linguistics. The middleware introduced in [28] considers a CPS comprised of physical systems generating aperiodic and periodic events that are processed on distributed computing platforms subject to end-to-end deadlines. The processing of a sequence of events forms a task, which are further divided into a chain of subtasks located on different processors. The middleware manages tasks through a set of components performing different services such as admission controller, idle resetter, load balancer, and task effector.

Similar event-based architectural and operational models of CPSs are presented in [25–27]. In such models, an event is a condition of interest and an action is a predefined operation following the detection of an event. Each event has attributes related to the occurrence of the condition, such as temporal and spatial properties. An occurrence of an event is detected by an observer which collects and processes data to check whether conditions are met. While the work in [25] focuses more on the definition of different types of events such as Physical Event, Physical Observation, Sensor Event, Cyber-Physical Event, and Cyber Event in the hierarchical event model, the models proposed in [26, 27] emphasize on how events are composed. For example, a concept-lattice-based composition method is proposed for composing CPS events from low-level physical events in [26], and an Adaptive Discrete Event (ADE) model is proposed in [27] that incorporates a Discrete Event Calculus

(DEC), rather than first order logic, to overcome inconsistencies in composition rules when composing events. Based on DEC, a set of reasoning rules is defined to build relationship between different events, which are then used to deduce output events. A common Smart Home example (automatic lighting) is used in [26, 27] to demonstrate the models.

3.2.2 Discussion

Events represent observations of the operating environment and interactions between different components of a CPS [24]. Event models are suitable for reactive or passive systems. However, they are insufficient for ERDCPSs for the following reasons:

- CPSs represented by event models in [25–27] are reactive systems, e.g, Smart Home. These systems have a main difference from ERDCPSs with respect to how an operation is triggered. Event models represent passive or reactive systems, where an operation is triggered when an event is detected, and, when no event occurs, the system stands by. ERDCPSs are systems that actively perform operations to fulfill a certain goal, and operations that lead to the goal are automatically triggered and executed. Thus, events and event-driven operations are not suitable for modeling operations of ERDCPSs.
- Another important feature of ERDCPSs missing in event models is environmental-resource-dependent operations. Operations defined in current existing event models are computing operations that process events and physical operations that react to events. These operations do not depend on environment (or external) resources.
- The operating environment of CPSs represented by event models includes only a single CPS. Cooperation and coordination with other CPSs is missing when designing and modeling such a CPS. However, the operating environment of an ERDCPS involves other peer ERDCPSs, which determines interactions between different ERDCPSs is an important feature that should be considered when modeling ERDCPSs.

3.3 Service-Based Models

Service-Based models [29–31] consider a CPS as an integration of components providing different services. Each service achieves a certain goal and operates under a certain context. An execution model of a CPS is comprised of different services that are tuned specifically for a certain operating context. To build a CPS operating in a particular environment, service specification and composition rules are required.

3.3.1 Overview

A context-sensitive and resource-explicit service model for CPSs is proposed in [29]. A CPS in such a model is comprised of various cyber and physical resources that are networked together. Each resource provides one or more services to achieve the desired goals under certain context and constraints. A context can be temporal, spatial or environmental. Service provision constraints specify limitations on how a service is provided. When a goal is set up for a CPS, service composition is performed. The composition process selects services to achieve the goal and specifies their operating contexts. PE-oriented (Physical Entity) service model [30] group physical entities with similar characteristics in a class hierarchy. Each physical entity inherits properties and services of its parent entity in the upper level. A PE-SOA (Service-Oriented Architecture) model is developed for PE and service specifications. A two-level composition approach is proposed to compose services. In the abstract level, a skeleton plan is obtained by composing services provided by physical entities that satisfies service provision constraints. In the physical level, the skeleton plan is refined to satisfy service context requirements.

One important part missing in [29, 30] is how to determine what services are required in order to achieve a task. Task workflow pattern, specification, and logic model are introduced in the service composition framework proposed in [31]. Workflow pattern is a sequence of PE states which are achieved in order to fulfill a task. Task specification describes a task in terms of PE states and specifies information such as names, inputs, participants, and

goals. Logic model establishes connections among tasks, PEs and services. The service composition process consists of two steps. First, given a task, a possible workflow pattern is selected. Then, the workflow pattern is instantiated by identifying a sequence of services that can connect the initial and goal state of the task. The searching for proper services is performed by AI planning.

3.3.2 Discussion

Service-based models focus on SOA design of CPSs based to enable a CPS adapt to different operating environments. One important feature of existing models is dynamic composition of services. The formal specifications of services enable autonomous composition when given a goal and a pool of services.

However, CPSs based on existing service models are loosely coupled in order to support dynamic composition that adapts to different operating environments. In this case, service models are similar to event-based models because they both represent passive or reactive systems, which is different from ERDCPSs. Moreover, the operating environment of a CPS defined by existing service models do not contains other peer CPSs that compete with each other for using resources.

Operations defined in these models are services to be provided, and resources defined in these models are computational and physical components of a CPS that provide services. However, as discussed before, operations of ERDCPSs depend on environment resources, which are external to any ERDCPS. Moreover, the connection between a service and physical operations are not clearly defined in existing models, and the service composition process concentrates on internal integration of a CPS, rather than interacting with the external environment.

3.4 Agent-Based Models

Agent-based CPS models [24, 33–35] consider a CPS or a component of a CPS as an agent in a distributed system. Agents interact and coordinate with each other to achieve a common goal. Each agent observes the operating environment and states of other agents, collects and interprets data, and makes decisions for next operations.

3.4.1 Overview

Two agent models are reviewed for modeling CPSs in [24]: autonomous agent and interactive agent. An autonomous agent consists of a knowledge base, a reasoner, a monitor, a learner, and a hardware abstraction layer. It interacts with the environment by sensing and affecting. An interactive agent (proposed in [33]) is formalized as an actor-like object which communicates by messages, interacts with the environment through interface points, and coordinates with other agents through policies. The distributed control of multiple CPSs is studied in [35], where each CPS is considered as an agent in a distributed system. The system model proposes a resilient control design for the multi-agent CPSs. A system framework that describes the interactions between Cyber and Physical components within a CPS, as well as the inter-dependency among multiple CPSs, are presented.

An agent model supports real-time decision in an information-rich environment, e.g., an intelligent *Water Distribution Network* (WDN) [34]. In such model, each agent represents an independent software component that manages physical resources within its local scope. An agent perceives the operating environment and collects data through sensors on a time- or event-triggered basis, and performs data integrity check and semantic interpretation. Semantic interpretation decides how collected data is semantically related to an agent through different semantic services. Management of physical commodities are decided through decision support algorithm based on data streams.

3.4.2 Discussion

Existing agent models view a CPS either as an agent being part of a larger system, or as a system consisting of different agents. Each agent operates independently, and interact with other agents and the operating environment. An agent can be considered as a single reactive system, and all agents together fulfill a common task or provide a service.

However, existing agent models focus more on internal operations such as data processing and interpreting, interactions between the cyber and physical subsystems, and the distributed control of different agents. Physical operations performed by a CPS are not clearly defined. Although interactions between agents are mentioned, they are confined to information sharing or control-feedback loop, which are different from interactions between ERDCPSs that compete for usage rights of environmental resources. Moreover, many features of ERDCPSs are absent in agent models, e.g., goal-driven and environmental-resource-dependent operations.

3.5 Resource Sharing

3.5.1 Real-Time Systems

Resource allocation or scheduling problem has been widely studied in Real-time Systems where computing resources, I/O devices, and communication channels are shared by different processes [48, 51, 52], and in database systems where data objects are shared by different applications [42–47]. The emergence of grid and cloud computing technologies also demands resource scheduling support to allocate computing resources or services to different applications or users [49, 50]. Many scheduling algorithms have been proposed to support different scenarios, such as fixed and dynamic priority scheduling where a process is given a priority to determine its precedence in competition for resources, soft real-time scheduling where tasks have no hard deadlines and job loss is permissible, and feedback scheduling where scheduling algorithms are adjusted dynamically depending on resource applicant's feedback.

Although resource scheduling algorithms are applied in different areas, they work similarly. Resources indicate either concrete or abstract objects that are used by different types of applicants such as processes, applications, and users. When the operation of an resource applicant requires one or more shared resources, the applicant first makes requests to a resource manager, and the resource manager decides whether to grant access rights of requested resources to the applicant and how long those requested resources can be used based on scheduling algorithms. Scheduling algorithms consider different properties of the applicant, such as priority, execution time length and deadline. They also consider the overall resource usage status of existing applicants. Based on the strategies applied by scheduling algorithms, resource requests are granted or rejected. If resources are granted, the scheduling process puts the applicant in a waiting queue to wait for its turn to use the resource. If resources are not available, the applicant is either blocked or aborted.

The resource sharing problem in ERDCPSs is similar to above traditional systems. For example, operations of an applicant (or ERDCPS) depends on external resources, and different applicants (or ERDCPSs) compete for requested resources. However, there exist some important differences.

- First, environmental resources in ERDCPSs are not scheduled or allocated purposely by any server (if there is any). More concern is placed on preventing resource usage conflicts, rather than maintaining a balanced usage load of a resource or ensuring fairness between applicants. Moreover, the resource scheduling process is transparent to the resource applicants in traditional real-time systems because of the central resource manager. However, a central resource manager may not exist in a CPS network, in which case, all ERDCPSs in the network have to coordinate with each other for resource usages to avoid conflicts.
- Second, in traditional systems, an applicant sends a request for resources, but it is the resource manager who decides when the resource can be used by an applicant. Although the applicant can specify how long it wishes to use the resource, it can't

decide the precise usage time of each resource. However, in ERDCPSs, the usage of an environmental resource by a physical operation is precise in both time and space. It specifies not only the resource that is needed, but also the time period during which the resource is used. For example, an increasing speed operation of a car in a particular position uses a set of road segments that can be calculated precisely (Figure 4.3) given the car’s current state, and the usage of these resources is specific in time and space. When an ERDCPS requests a resource for its operation, it specifies both the resource and the usage time. So not only the availability of the resource, but also the availability of the specified usage times are checked. Compared to resource applicants in traditional systems, an ERDCPS has a higher level of autonomy over the resources.

- Third, in most cases, when an applicant can’t get requested resources, it is blocked, and waits for its turn to use the resources. For example, an application can wait for its turn to use I/O devices to read from or write to files. However, this is not possible for physical operations of an ERDCPS. Because physical operations are real-time and an ERDCPS keeps doing physical operations to maintain its state in a CPS Network. If no physical operations are executed, an ERDCPS will be placed in a inconsistent state which causes bad consequences such as car collisions.

3.5.2 Traffic Management Systems

Resource sharing that is sensitive in time and space has been studied in traffic management systems [53, 54, 57]. A hierarchical decentralized planning framework is studied in [54]. In this framework, the flight path is represented by a graph, and different flight share the same space for their movements. In the hierarchical decision-making structure, different decision makers in the middle level file the path planning of their airplanes (in the bottom level) to FAA (Federal Aviation Administration) on the top level, where the plane paths are finally decided. The intersection management system proposed in [53] considers a road intersection as a set of grids that are used by a vehicle to finish a turning or moving operation.

Although resource sharing in these systems is similar to that of ERDCPSs with respect to the sensitivity of time and space, there are some differences. First, similar to traditional real-time systems, a central site is used to save the resource usage information for conflict detection and resource allocation. But in ERDCPSs, we may not have such a central site and more focus is placed on avoiding conflicts. Second, the resource sharing algorithm discussed is domain-restricted and can be used only in some particular applications such as intersection and air traffic management. We are seeking to build a general model for all ERDCPSs that can be applied to different application areas. Last, the connection between resources and operations is not clearly defined, which is one of the goals we should achieve when building a model for ERDCPSs.

3.6 Candidate: Transaction Model

Transactions are used to model operations of a database system (DBS) [38–40]. A transaction consists of read and write operations from clients to access data objects (which are considered as resources). The state of a data object is represented by its value. A read operation does not change the state of a data object, but a write operation does. To prevent data inconsistency, such as lost update and inconsistent retrieval ([41]) caused by concurrent access to data objects from different clients, concurrency control algorithms are required to process transactions. Different concurrency control strategies ([42–47]) have been developed such as two-phase locking, timestamp ordering, optimistic nonlocking, and nested transactions.

Similarities between a DBS and an ERDCPS have motivated us to consider using transactions to model ERDCPS.

- Resource-dependent operations. Read and write operations of a client relies on data objects they access, as operations of an ERDCPS rely on environmental resources.
- Exclusive access. When changing the state of an data object (write operation), exclusive access to the data object is required. A physical operation of an ERDCPS that

creates physical effects to the physical world requires exclusive access to environmental resources when it is being performed.

- External resources. From the perspective of a client, data objects are external resources and beyond its control, as environmental resources are external to an ERDCPS. Management of data objects will be similar to management of environmental resources if a central resource manager exists in a CPS Network.
- Multiple users. There exist multiple clients that share the same database, as multiple ERDCPSs share the same set of environment sources. For this reason, concurrency control protocols are required to coordinate different clients' operations to avoid conflicts, which are also required for different ERDCPSs.

These similarities enable adoption of transaction models and algorithms of a DBS in modeling ERDCPSs. However, there are some significant differences between a DBS and an ERDCPS that determine that transaction models of a DBS can't be directly used for an ERDCPS.

- Sensitivity in time and space. Operations of a DBS are insensitive to time and space. Read and write operations of clients don't specify when and which copy of a data object is accessed. These are decided by the server. Resource access is controlled by the server, which introduces a layer of uncertainty and makes usage of resources by clients unpredictable. However, operations of an ERDCPS themselves precisely define the time and location of a resource access.
- Paramertized operations. Operations of an ERDCPS are more complex than those of a traditional DBS, which are either read or write. For example, a car can turn right or left, move forward and backward. Each type of operation has some attributes that determine specific behavior of an operation, e.g., turning operation has angle and time as attributes (presented later in model demonstration). But for a traditional DBS, references and values of data objects are what are needed for read and write

operations. Moreover, the complexity in operations of an ERDCPS results in more complex composition of a transaction.

- Resource flexibility. In a DBS, when read and write operations refer to a data object, only that data object can guarantee correct computation of a client and the server cannot provide another data object as a substitute to the client. However, in an ERDCPS, the same operation (with different start states) can be performed with different sets of environmental resources. For example, road segments required by a turn right operation varies with the starting position of a car. A turn right operation at point A and point B requires different resources. Thus, when the required resources can't be obtained by an ERDCPS operation, alternative resources can be considered if the start state can be adjusted.
- Atomicity. Because of real-time property of an ERDCPS, when some physical operation fails, the effect of its partial execution cannot be undone and the resulting state cannot be rolled back to its starting state. The next operation has to operate based on the context created by the previous operation, partially or completely executed. When a DBS operation fails, however, effects caused by the partially executed operation can be reversed or compensated to restore involved data objects back to their start states in order to prevent inconsistency. The next operation is guaranteed to start with a consistent start state.
- Resource Management. A DBS achieves a good control of concurrent data access operations because of the management layer, and the coordination of resource usages is transparent to the clients. However, the operation environment of an ERDCPS is distributed in nature, and there may not exist such a central entity to manage access to environmental resources. Thus, an ERDCPS needs to work with other ERDCPSs to coordinate usages of environmental resources to prevent inconsistency and avoid conflicts.

3.6.1 Summary

Figure 3.1 lists features of the operating environment, system interaction, and operations of each type of existing model and ERDCPSs.

Feature	Operating Environment		System Interaction		Operations				
	Composition	Example	Internal	External	Operation	Active or Passive	Properties	Context	External Resource
Event Model	single CPS, environment	Smart Home, iLight	event interpretation	event detection	physical operation	passive	time and space of an event, conditions	event or condition	no
Service Model	single CPS, environment	earthquake rescue mission, safety guard	service composition	event, goal, and context driven	tuned service, physical operation	passive	provision constraints, operating context, operation effect	goal, context	no
Agent Model	multiple agents, environment	water distribution network	data collection and interpretation	event detection	physical operation	passive	conditions, data	event or condition	no
Resource Allocation	multiple clients, a server, shared resources	communication channel, computing grid	resource allocation	resource access requests	resource usage	client: active server: passive	resource usage status, time length, attempt count	program state	computing time, I/O devices, etc.
Database System	multiple clients, a server, databases	database system	read/write locks	data access requests	read or write data objects	client: active server: passive	data objects and values	database state	data objects
ERDCPS	multiple ERDCPSs, shared resources	autonomous cars, warehouse robots, assembly line robots	operation processing	a given goal or coordination requests	physical operation, resource usage	active	physical properties, time, space and resource	system state	environmental resource

Figure 3.1: Comparison of Existing Models and ERDCPSs

Chapter 4

A Transaction Model for CPS

4.1 Introduction

From Chapter 2 and 3, we know that existing models are not able to capture the characteristics of an ERDCPS and its operating environment. In this chapter, we propose a new transaction model for an ERDCPS. We first give a sample transaction of an autonomous car, which is presented in Chapter 2 as an ERDCPS example. Then we propose the definitions of a Cyber-Physical System and its operating environment *CPS Network* with respect to ERDCPSs. After that, we define the concepts of an action and a transaction that provide different levels of abstractions for physical operations. The resource-dependent feature of a physical operation is defined in action and transaction. At last, we define the schedule of a CPS Network, which consists of transactions from all CPS entities, and the compatibility between two transactions with respect to resource usages.

4.2 A Sample Transaction

Before presenting our model, we give a sample transaction of the autonomous car example mentioned in Chapter 2. Figure 4.1 shows an instance of transaction *ChangeToRightLane* of an autonomous car. We'll use this transaction example to present definitions of our transaction model. The default process of *ChangeToRightLane* is:

1. Turn on indicator signaling that a right turn is going to occur;

2. Turn the wheel to right (clockwise) with 30 degree and keep that angle for 2 seconds to let the car move into the right lane;
3. Turn the wheel to left (anti-clockwise) with 30 degree and keep that angle for 1 second to adjust the car's moving direction to straight ahead;
4. Turn off the indicator.

```

Transaction: ChangeToRightLane
Start Context: System State  $S_s$ 
Start
    IncreaseSpeed(10 m/s, 3s) //temporary action
    TurnOnIndicator("right")
    TurnRight(30 d)
    TurnLeft(30 d)
    TurnOffIndicator("right")
End
(m - meter, s - second, d - degree)

```

Figure 4.1: An Instance of Transaction ChangeToRightLane

All steps (except *IncreaseSpeed(10 m/s, 3s)*) are **inherent** actions of *ChangeToRightLane*. *IncreaseSpeed* is a **temporary** action interpolated to adjust the start system state (the need for this is defined later).

4.3 CPS and CPS Network

Definition 4.3.1 (Cyber-Physical System). A Cyber-Physical System (CPS) consists of a physical (*PhyS*) and a computational (*ComS*) subsystem, as shown in Figure 4.2. *PhyS* is composed of physical components such as communication devices, sensors, actuators, and mechanical systems. *ComS* is composed of software components such as Goal Manager (GM), Transaction Manager (TM), and Resource Manager (RM) performing different functions.

ComS works as a planner and *PhyS* works as an executor. They are highly coupled in the following ways:

- *ComS* does computations to find solutions for problems confronting *PhyS*, and instructs *PhyS* to perform proper physical operations;
- *PhyS* follows instructions from *ComS*, performs expected operations, and gives feedback to *ComS* to enable status tracking and operation planning.

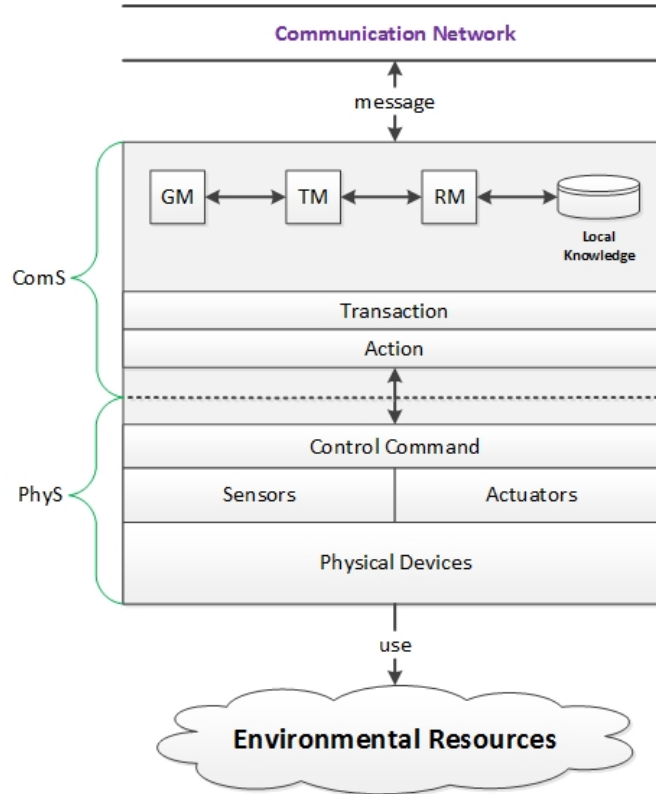


Figure 4.2: Cyber-Physical System

Definition 4.3.2 (Environmental Resources). Environmental resources represent objects in the operating environment that a CPS relies on to perform physical operations, such as road segments required by a car to make movements. Environmental resources are external to and beyond the control of a CPS. We use **ER** to represent the set of environmental resources.

In our car example, a grid map is used to represent a road. A road is divided into a two-dimensional map of grids with each grid representing a fixed length road segment in one

lane of the road. A grid is identified by its column and row indices (x and y) in the map where x indicates the lane in which the road segment lies and y indicates the road segment within lane x . Thus, \mathbf{ER} is a set of grids.

$$\mathbf{ER} = \{(r, x, y) \mid (r, x, y) \text{ represents road segment } y \text{ of lane } x \text{ in road } r.\}$$

Definition 4.3.3 (CPS Network). A CPS Network (Figure 1.2) is a collection of CPSs that are interconnected through a network (wired or wireless) and interact with each other by sharing environmental resources. Each CPS is called a *CPS Entity*.

Definition 4.3.4 (System State). Each CPS entity has a **system state** indicating its status in a CPS Network at a particular time. For example, a car's system state at time t is defined below.

$$\mathbf{CarState}^t = (\text{speed}, \text{acceleration}, \text{road}, \text{lane}, \text{position})$$

Here,

- t indicates the time when a system state is sampled.
- road , lane , and position give a car's location in a grid map at time t .

CPS entities in a CPS Network share the same set of environmental resources. Interactions among these entities occur due to competition for *usage times* of shared resources in order to perform physical operations. Physical operations of a CPS entity require exclusive access to certain resources during its usage time. If different CPS entities access the same resource at the same time, a resource usage conflict occurs.

The state of a CPS Network $\mathbf{CNState}^t$ indicates the usage status of environmental resources by all CPS entities at a particular time t .

$$\mathbf{CNState}^t = \{(Res, E) \mid Res \subseteq \mathbf{ER}, E \text{ is a CPS entity}\}$$

The pair (Res, E) indicates that a CPS entity E is using resources indicated by a set Res at time t .

Definition 4.3.5 (State Consistency of CPS Network). A CPS Network is in a *consistent* state at a given time *iff* (if and only if) usages of environmental resources by all CPS entities have no conflicts at that time. Thus, $\mathbf{CNState}^t$ is consistent *iff*

$$\forall (Res_i, E_i), (Res_j, E_j) \in \mathbf{CNState}^t \ (i \neq j), Res_i \cap Res_j = \emptyset$$

4.4 Actions

Definition 4.4.1 (Action). An action is composed of a sequence of control commands from device drivers (Figure 4.2) and provides an abstraction of physical operations performed by *PhyS* to create physical effects, such as giving signals through actuators (e.g., *TurnOnIndicator("right")*), and manipulating mechanical system to do movements (e.g., *IncreaseSpeed(10, 3)* and *TurnRight(30)*). Actions are part of the interface between *ComS* and *PhyS* that enables *ComS* to control *PhyS*, and further to interact with environmental resources and other CPS entities in a CPS Network.

Depending on the physical effects an action can create, actions are categorized into different *types*, such as *TurnRight*, *TurnLeft*, and *IncreaseSpeed*. Each type of action represents a particular pattern of physical operations performed by a CPS. An **action type** is represented as:

$$ActionType = (\mathbf{Par}, \mathbf{Pre}, \mathbf{Post}).$$

Here, **Par** is a set of attributes that determine specific physical behavior of the action, and **Pre** and **Post** are sets of preconditions and postconditions respectively.

For example, **Par** of *TurnRight* action is $\{turningAngle\}$, and **Par** of *IncreaseSpeed* action is $\{targetSpeed, timeDuration\}$.

By giving values to attributes in **Par**, an *action instance* is created, and the process is called *action instantiation*. We use the following form to represent an **action instance** (assume that it has n attributes):

$$Action (value_1, value_2, \dots, value_n)$$

Here, $value_i$ is the value for $attribute_i$. For example, $TurnRight(30)$ is an instance of action type $TurnRight$, and $IncreaseSpeed(10, 3)$ instance of type $IncreaseSpeed$.

The execution of an action is carried out in a system state of a CPS. Specifically, system states right before and after executing an action are called **start** and **end system state** of the action, represented by S_s and S_e respectively. Consider a CPS as a state machine, an action causes state transition from S_s to S_e :

$$S_s \longrightarrow_a S_e, (a \text{ is an action}),$$

Pre and **Post** specify conditions that must be satisfied by S_s and S_e respectively. Assume that action $IncreaseSpeed(target, duration)$ is being executed: starts at time t_s and ends at time t_e . The state corresponding to t_s and t_e are S_s and S_e , both of type **CarState**. Each car has a maximum speed ($MaxSpd$) and a maximum acceleration ($MaxAcc$) that it can achieve. **Pre** of $IncreaseSpeed$ will be:

$$\{ (S_s.speed < target \leq MaxSpd), (\frac{target - S_s.speed}{duration} \leq MaxAcc) \}.$$

The first condition indicates that the target speed should be larger than the current speed and not exceeding the maximum speed, and the second indicates that the required acceleration not exceeding the maximum acceleration. **Post** of $IncreaseSpeed$ will be:

$$\{ (S_e.speed == target), (t_e - t_s \leq duration) \}.$$

The first condition says after execution of the action, the speed will reach the target speed, and the second means the time taken to increase the speed is not more than the time allowed.

Under a given start system state, the execution of an action by $Phys$ may require environmental resources in order to perform its physical operations. For example, road segments that a car has moved on when doing actions $IncreaseSpeed(10,3)$ or $TurnRight(30)$ are environmental resources it needs for both actions. Figure 4.3 shows resource usages of $Car I$ in a two-lane road. The shaded part in Lane l_1 indicates road segments required by an $IncreaseSpeed$ action and the shaded part in Lane l_2 by a $TurnRight$ action. However,

not all actions require environmental resources for execution. For example, we assume the execution of action *TurnOnIndicator* starts and finishes instantly, and thus no environmental resources are required.

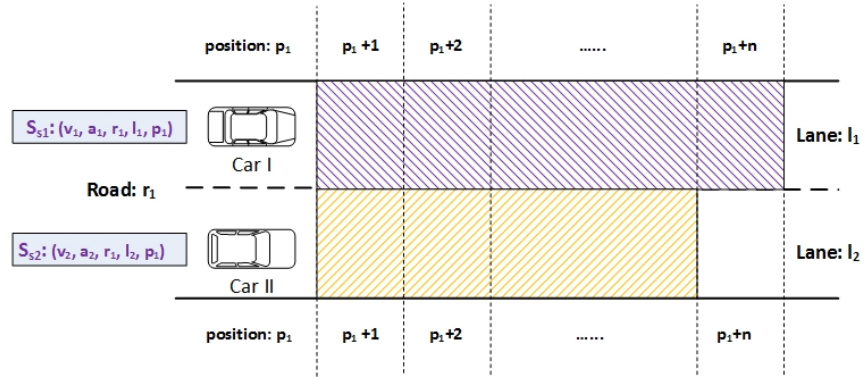


Figure 4.3: Usage of Environmental Resources by Actions

Definition 4.4.2 (Action Resource Usage Set). The **resource usage set** of an action a , represented by \mathbf{UResr}_{a,S_s} , indicates usages of environmental resources when executing an action a in state S_s . Each element in \mathbf{UResr}_{a,S_s} is a pair containing a resource res and a time period $[t_s, t_e)$ during which res is used by a .

$$\mathbf{UResr}_{a,S_s} = \{(res, [t_s, t_e)) \mid res \in \mathbf{ER}, t_s < t_e\}.$$

\mathbf{UResr}_{a,S_s} can be calculated given attribute values of a and S_s . For example, assume that a grid map is used to represent a road. Assume that *Car I* (Figure 4.3) starts executing action *IncreaseSpeed(target, duration)* at time t_1 under a start system state S_{s1} : $(v_1, a_1, r_1, l_1, p_1)$. Then \mathbf{UResr} of *IncreaseSpeed(target, duration)* is given below:

$$\begin{aligned} & \{((r_1, l_1, p_1 + 1), [t_1, t_1 + 1)), \\ & ((r_1, l_1, p_1 + 2), [t_1 + 1, t_1 + 2)), \\ & \dots, \\ & ((r_1, l_1, p_1 + \lceil (v_1 t + at^2/2)/l \rceil), [t_1 + (t - 1), t_1 + t))\}. \end{aligned}$$

Here,

- l is the fixed length of a road segment.
- t equals *duration* and is the time taken to increase the speed.
- a is the minimum acceleration required during the increasing process: $(target - v_1)/duration$.
- $(r_1, l_1, p_1 + i)$ ($1 \leq i \leq \lceil (v_1 t + at^2/2)/l \rceil$) represents grid i relative to the car's initial location (r_1, l_1, p_1) (grid 0).
- $[t_1 + (j - 1), t_1 + j)$ ($1 \leq j \leq duration$) is the usage time period (1 second) of the resource $(r_1, l_1, p_1 + \lceil (v_1 j + aj^2/2)/l \rceil)$.

When execution of *IncreaseSpeed(target, duration)* is completed, the end system state S_e (at time $t_1 + t$) would be:

$$(target, a_1, r_1, l_1, p_1 + \lceil (v_1 t + at^2/2)/l \rceil).$$

From the above example, we can see how action attribute values (*target* and *duration*) and start state (S_s) determine the physical effect of an action, i.e., resource usage \mathbf{UResr}_{a,S_s} and end system state S_e . On the one hand, for an action a which has fixed attribute values (*target* and *duration* in the example), \mathbf{UResr}_{a,S_s} and S_e is decided by start system state S_s . Different start states give different resource usage sets and end system states. On the other hand, given the same start state S_s , different actions of the same type, which have different attribute values, have different resource usage sets and end system states. As we would discuss later, these two properties enable *dynamic adjustment* of a transaction to change its resource usages and avoid transaction conflicts.

In the resource set example above, environmental resources are represented in an *absolute* way, e.g., (r_1, l_1, p_1) , or grid 0 of lane l_1 on road r_1 . There is another way to represent environmental resources: relative. In the *relative* approach, resources are specified relative to a start state S_s . For example, grid i relative to S_s is given by $(S_s.road, S_s.lane, S_s.position +$

$i * l$). The relative approach allows a more flexible specification of resource usage, e.g., Car I can require resources such as “10 meters ahead of *Car II*” for its *TurnRight* action. However, no matter which way of resource representation is used, CPS entities in a CPS Network must share the same notion of environmental resources in order to understand each other when negotiating about resource usages.

4.5 Transactions

Definition 4.5.1 (Transaction). A transaction represents a sequence of actions $a_1, a_2, a_3, \dots, a_n$ ($n > 0$), which together fulfill a certain task such as changing a lane, constant-speed moving, accelerating, and decelerating. For instance, in Figure 4.1, the sequence of transaction *ChangeToRightLane* includes actions: *IncreaseSpeed(10,3)*, *TurnOnIndicator(“right”)*, *TurnRight(30)*, *TurnLeft(30)*, and *TurnOffIndicator(“right”)*.

A transaction provides a higher level abstraction of physical operations than an action (Figure 4.2). It groups different actions into an execution sequence and represents the process carried out by a CPS in order to complete a certain task. A transaction is the unit of operation managed by *ComS* to control operations of *PhyS*. A goal to be achieved by a CPS is decomposed into a sequence of tasks with each one achieved by a transaction.

Similar to actions, transactions also have different *types*. For example, transaction types of an autonomous car when moving on a highway include: *ChangeToLeftLane*, *ChangeToRightLane*, *Accelerate*, *Decelerate*, *MakeRightTurn*, *MakeLeftTurn*, *ConstantSpeedMove*, etc. Each type of transaction defines a default composition of actions of different types. A **transaction type** is represented as follows:

$$\mathbf{TransactionType} = (\mathbf{Act}, \mathbf{Pre}, \mathbf{Post}).$$

Here,

- **Act** is an ordered sequence of action types. When a transaction is executed, actions are executed sequentially according to the predefined order.

- **Pre** and **Post** are sets of preconditions and postconditions that specify conditions that must be satisfied on entry to or exit from execution of a transaction.

By instantiating each action type in **Act**, a transaction type is instantiated and a corresponding *transaction instance* is created. For example, the transaction in Figure 4.1 is an instance of transaction type *ChangeToRightLane*. Actions instantiated from action types predefined in the action sequence are called **inherent** actions of the transaction, e.g., actions except *IncreaseSpeed(10,3)* in Figure 4.1. Otherwise, they are **temporary** actions, e.g., *IncreaseSpeed(10,3)* in Figure 4.1. Temporary actions are added into the predefined sequence of actions to adjust the transaction. We use the following form to represent a **transaction instance**:

$$Transaction (action_1, action_2, \dots, action_n),$$

where $action_i$ is an action instance instantiated from its corresponding action type.

Similar to actions, we use S_s and S_e to represent system states right before and after executing a transaction. A transaction T causes state transition from S_s to S_e :

$$S_s \longrightarrow_T S_e$$

Preconditions specify requirements on S_s in order to execute a transaction, e.g., a *ChangeToRightLane* transaction requires that a car is currently not in the rightmost lane of the road. Postconditions describe what S_e will be after the successful execution of a transaction, e.g., when a *ChangeToRightLane* transaction is completed, we have $S_e.lane == S_s.lane + 1$.

Definition 4.5.2 (Transaction Resource Usage Set). \mathbf{UResr}_{T,S_s} indicates usages of environmental resources when executing a transaction T with a start system state S_s . It is a combination of resource usage sets of its actions.

$$\mathbf{UResr}_{T,S_s} = \bigcup_{i=1}^n \mathbf{UResr}_{a_i,S_{si}}.$$

Here, a_i ($1 \leq i \leq n$) indicates the i_{th} action of T and S_{si} is its start system state.

Given S_s and the profile (parameters, pre and post conditions) of each action of T , a sequential deduction can be made to obtain the start (S_{si}) and end (S_{ei}) system state of each action a_i ($1 \leq i \leq n$). The process is shown below:

$$\begin{aligned}
S_s \Rightarrow S_{s1} \xrightarrow{a_1} S_{e1} \Rightarrow S_{s2} \xrightarrow{a_2} S_{e2} \Rightarrow S_{s3} \xrightarrow{\quad} \\
\quad \dots \xrightarrow{\quad} S_{e(n-1)} \Rightarrow S_{sn} \xrightarrow{a_n} S_{en} \Rightarrow S_e
\end{aligned}$$

Since actions of a transaction are executed sequentially, we have:

- S_s will be the start system state of a_1 , S_{s1} ,
- The end system state S_{en} of a_n will be S_e ,
- The end system state of an action will become the start state of next action in the sequence, e.g., $S_{s2}=S_{e1}$.

To make changes to \mathbf{UResr}_{T,S_s} , we can simply change resource usage sets of its actions $\mathbf{UResr}_{a_i,S_{si}}$ by adjusting their attributes values and start system states. The above process indicates S_{si} ($1 \leq i \leq n$) are determined by S_s and the sequence of actions. By adjusting S_s and the action sequence, we can change S_{si} and $\mathbf{UResr}_{a_i,S_{si}}$, and further change \mathbf{UResr}_{T,S_s} . This provide us two ways to adjust transaction resource usage set dynamically: action adjustment and sequence adjustment. Action adjustment focuses on changing action attribute values and start system state, while sequence adjustment focus on changing the sequence of actions of a transaction. Details will be covered later when we present transaction processing algorithms.

4.6 Schedules

We have so far only considered operations (actions and transactions) of a single CPS. In a CPS, a transaction is the management unit applied by $ComS$ to control operations of $PhyS$. When $ComS$ schedules a transaction, it is sent to $PhyS$ for execution. At any time, $PhyS$

can only execute one transaction. Thus, transactions of a CPS are executed sequentially by its physical subsystem.

In a CPS Network, there are multiple CPSs operating at the same time and operations of one CPS may interfere with another when they share environmental resources. Transactions scheduled for execution by different CPSs constitute a schedule and they are executed separately and concurrently by different CPSs. Physical operations of a CPS entity require exclusive access to certain resources during its usage time. For example, if two cars use the same road segment at the same time, a collision occurs.

Definition 4.6.1 (Compatible Transactions). Two transactions are *compatible* if they have no conflicts. A **conflict** occurs between two transactions, T_i and T_j , when they use the same environmental resource at the same time, i.e.,

$$\begin{aligned} & \exists(res_i, [t_{si}, t_{ei})) \in \mathbf{UResr}_{T_i, S_{si}}(t_{si} < t_{ei}), \exists(res_j, [t_{sj}, t_{ej})) \in \mathbf{UResr}_{T_j, S_{sj}}(t_{sj} < t_{ej}), \\ & res_i == res_j \text{ and } (t_{si} \leq t_{sj} < t_{ei} \text{ or } t_{si} \leq t_{ej} < t_{ei}). \end{aligned}$$

We use notation $\bigcap_{overlap}$ to represent the operation of finding overlaps between resource usage time periods of two transactions for the same resource. So, T_i and T_j are **compatible** iff

$$\mathbf{UResr}_{T_i, S_{si}} \bigcap_{overlap} \mathbf{UResr}_{T_j, S_{sj}} = \emptyset.$$

Since the resource usage set of a transaction is a combination of usage sets of its actions, the equation above is equivalent to

$$\forall a_k \in \mathbf{Act}_{T_i}, \forall a_l \in \mathbf{Act}_{T_j}, \mathbf{UResr}_{a_k, S_{sk}} \bigcap_{overlap} \mathbf{UResr}_{a_l, S_{sl}} = \emptyset.$$

If no conflicts are found between T_i and T_j , concurrent execution of them is allowed. Otherwise, measures are needed to resolve conflicts before they can be executed.

Definition 4.6.2 (Schedule). A schedule H is an execution sequence of actions from a set of transactions **TS**: $\{T_1, T_2, T_3, \dots, T_n\}$ ($n > 0$) that belong to CPS entities in a CPS Network. It has following properties:

- Transactions from the same CPS entity are executed serially and transactions from different CPS entities are executed concurrently.
- Transactions in **TS** share the same set of environmental resources to perform their operations and conflicts may occur.

If all transactions in **TS** come from the same CPS entity, then the schedule preserves consistency of a CPS Network because all transactions are executed serially and use environmental resources at different times. A schedule with transactions coming from different CPS entities, however, may not preserve consistency because concurrent execution of transactions may have conflicts.

Definition 4.6.3 (Consistent Schedule). A schedule is *consistent* iff

$$\forall T_i, T_j \in \mathbf{TS} (i \neq j), \mathbf{UResr}_{T_i, S_{s_i}} \bigcap_{\text{overlap}} \mathbf{UResr}_{T_j, S_{s_j}} = \emptyset.$$

The formula states that if transactions in a schedule are compatible with each other, then the schedule is consistent. A consistent schedule preserves consistency of a CPS Network.

4.7 Summary

In this chapter, we propose our transaction model for ERDCPSs. In this model, the operating environment of a CPS is represented by a CPS Network, where Environmental Resources, CPS entities, and a Communication Network are grouped together. The environmental resources are shared by CPS entities to perform their physical operations.

Physical operations of a CPS entity are abstracted into two levels: action and transaction. An action represents a sequence of control commands upon the physical subsystem, such as *TurnRight* and *IncreaseSpeed* actions for an autonomous car. A transaction represents a sequence of actions. For example, a *ChangeToRightLane* transaction consists of four actions: *TurnOnIndicator*, *TurnRight*, *TurnLeft*, and *TurnOffIndicator*.

Transactions of all CPS entities in a CPS Network constitutes a schedule. Since environmental resources are shared, resource usage conflicts may exist between transactions of different CPS entities. Two transactions are compatible with each other if they don't have conflicting resource usages. To make a schedule consistent on using environment resources, all its transactions must be compatible with each other.

We only present the definitions of the transaction model in this chapter. The transaction processing algorithm will be presented in Chapter 5, where we show how to maintain a consistent schedule in a CPS Network. The implementations of the model and algorithms will be presented in Chapter 6.

Chapter 5

Transaction Processing

5.1 Introduction

We present the definition part of the transaction model in the previous chapter, and, in this chapter, we continue with the algorithm part: transaction processing. To guarantee the successful execution of a transaction of a CPS entity, the transaction processing procedure should avoid resource usage conflicts between transactions. Each CPS entity in the network must do its job to detect and resolve conflicts in order to maintain a consistent schedule of transactions.

We first give an overview of the two-phase commit transaction processing algorithm. The two-phase commit algorithm utilizes the fact that physical operations of an ERDCPS follow certain patterns or rules, which makes resource usages of a transaction predictable given its start system state. A transaction pre-Write set represents the predicted resource usages of a transaction and is used to detect potential transaction conflicts.

In the pre-commit phase, a transaction is scheduled by reserving the pre-Write set. The scheduling process detects and resolves potential transaction conflicts. Several strategies, e.g., Win-Lose, Win-Win, and Transaction Preemption, are proposed to resolve different types of conflicts. Dynamic transaction adjustment is introduced to show how a transaction can be adjusted in order to resolve potential conflicts. Two general transaction processing algorithms are presented for the centralized and distributed resource management environ-

ments respectively.

In commit phase, a transaction is executed by the physical subsystem and the usage of a resource is supposed to follow what has been predicted in the pre-commit phase. An exception occurs when unexpected resource usage happens. An exception is caused either internally or externally. We define internal and external exceptions, and discuss how both types of exceptions are handled in the commit phase.

5.2 Two-Phase Commit Transaction Processing

Conflicts between transactions prevent their successful execution. Rather than detecting conflicts when executing a transaction in real-time, an alternative strategy is to check potential conflicts when scheduling a transaction. Potential conflicts between two transactions can be identified if we know their potential resource usages. From previous sections, we know that resource usages of a transaction depend on the start execution time and start system state, which are the end execution time and end system state of the most recent executed transaction from the same CPS entity.

5.2.1 Transaction pre-Write Set

A transaction *pre-Write* set, $\mathbf{preUResr}_{T,S_s}$, is a predicted resource usage set of a transaction T under start system state S_s . The only difference between $\mathbf{preUResr}_{T,S_s}$ and \mathbf{UResr}_{T,S_s} is that resource usages of the former are predicted, while those of the latter actually occur when executing T . Thus, \mathbf{UResr}_{T,S_s} is the *Write* set of T .

Similar concepts can be applied to an action, so $\mathbf{preUResr}_{a,S_s}$ and \mathbf{UResr}_{a,S_s} are the *pre-Write* and *Write* sets of an action a with respect to a start system state S_s .

Assume that T contains a sequence of actions: a_1, a_2, \dots, a_n ($1 \leq i \leq n$), then

$$\mathbf{preUResr}_{T,S_s} = \bigcup \mathbf{preUResr}_{a_i,S_{s_i}}, a_i \in \mathbf{Act}_T.$$

For any two transactions T_i and T_j , if we have their pre-Write sets ($\mathbf{preUResr}_{T_i,S_{s_i}}$ and $\mathbf{preUResr}_{T_j,S_{s_j}}$), then we are able to detect potential conflicts by checking whether they

are compatible using their pre-Write sets. If

$$\mathbf{preUResr}_{T_i, S_{s_i}} \bigcap_{\text{overlap}} \mathbf{preUResr}_{T_j, S_{s_j}} = \emptyset,$$

then no potential conflicts exist between them.

5.2.2 Two-Phase Commit

Two-phase commit transaction processing mechanism is based on the concept of transaction pre-Write set. When a transaction is triggered, we can get its expected start execution time and expected start system state from the previous scheduled transaction, and calculate its pre-Write set.

The **pre-commit** phase indicates scheduling a transaction. The main task of the pre-commit phase is to check whether potential conflicts exist between T and transactions from other CPS entities, and reserve environmental resources for execution of T . The resource required and their usage times by T are indicated by $\mathbf{preUResr}_{T, S_s}$. The scheduling process includes the following steps:

1. Get necessary information of T , which includes expected start and end execution times, expected start and end system states (S_s and S_e), and pre-Write set ($\mathbf{preUResr}_{T, S_s}$).
2. Send a reservation request to the resource server or other CPS entities in a CPS Network to reserve $\mathbf{preUResr}_{T, S_s}$. Each recipient of the request checks whether $\mathbf{preUResr}_{T, S_s}$ will cause conflicts with their transactions and sends corresponding responses back.
3. Collect responses and resolve potential conflicts if any.
4. Confirm the reservation if successful; Otherwise, abort T .

When a successful reservation is made, $\mathbf{preUResr}_{T, S_s}$ is reserved, and no other transactions are supposed to use reserved resources at time periods reserved by T . In this case,

T is said to be **pre-committed**. If reservation doesn't succeed, T can not be executed without causing conflicts and thus it is aborted. $ComS$ has to seek alternative transactions to replace T that will not cause conflicts with other transactions.

The **commit** phase indicates executing a transaction. The execution of T is expected to follow what is predicted, i.e., use only specified resources in reserved time periods. An exception occurs when any difference from what is expected of resource usages is found. Exceptions can cause transaction conflicts in realtime because either usage of unreserved resources by a transaction. To prevent exceptions, the execution of T is monitored by $ComS$ and realtime resource usages are checked against predicted ones in $\mathbf{preUResr}_{T,S_s}$. When potential exceptions are found, they are handled immediately by $ComS$. If exceptions can't be handled in time and cause realtime conflicts, $ComS$ takes actions to reduce undesired effects.

The reservation of transaction pre-Write set can be considered as implicit locking. A successful reservation in pre-commit phase indicates that certain access time periods of specified resources are locked for a transaction, which prevents other transactions from using these resources at reserved times. When commit phase finishes, these locks are released automatically because reserved access times has passed and corresponding locks vanishes with the passing time. To make two-phase commit mechanism work, all CPS entities in a CPS Network should follow the same protocol to process transactions.

5.2.3 Transaction State Machine

Given the two-phase commit processing mechanism described above, a transaction can be in six different states (Figure 5.1): Triggered, Scheduling, Aborted, PreCommitted, Executing, and Committed (Completed or Incomplete).

1. When a transaction T is triggered, it enters **Triggered** state.
2. When T is being scheduled, it is in **Scheduling** state.

3. There are three possible states reachable from *Scheduling* state. When scheduling is successful, T becomes **PreCommitted**. When potential conflicts are found, T may be adjusted to solve conflicts. This results in a self-transition of **Scheduling** state. If scheduling is unsuccessful (conflicts can not be solved), T is aborted and enters **Aborted** state.
4. When T is pre-committed, it is ready for execution. However, there are cases where pre-committed T can be canceled, e.g., potential exceptions are found before it is executed. These cases result in abortion of T . If no abortion occurs, T can be executed. When the execution starts, T is in **Executing** state.
5. When the execution terminates, T is in **Committed** state and T can be either completely executed (sub-state **Complete**) or partially executed (sub-state **Incomplete**).

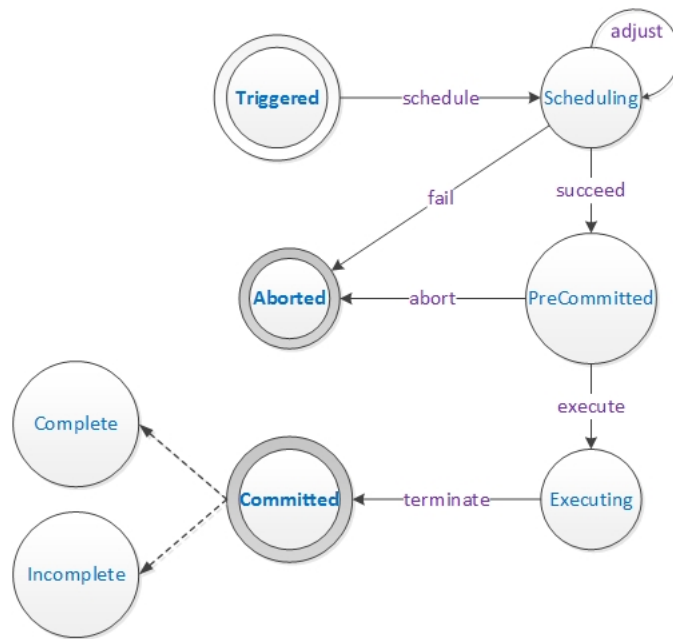


Figure 5.1: Transaction State Machine

5.2.4 Action-Level Two-Phase Commit

Different from above **transaction-level** two-phase commit mechanism, an alternative is **action-level** two-phase commit, where pre-Write sets are computed and reserved action by action, rather than transaction by transaction. In action-level scenario, T is executed without $\mathbf{preUResr}_{T,S_s}$ being reserved first. Whenever an action a_i of T that requires environmental resources is going to be executed, it is processed in a two-phase commit manner. The process is similar to transaction-level two-phase commit:

1. First, get S_{s_i} and compute $\mathbf{preUResr}_{a_i,S_{s_i}}$.
2. Then send out a request to reserve $\mathbf{preUResr}_{a_i,S_{s_i}}$. When requested resources are reserved for a_i , a_i is pre-committed and is ready to be executed.
3. Finally, a_i is sent to $PhyS$ for execution in the commit phase, and when execution is terminated, it is committed.

In action-level two-phase commit processing mechanism, execution of T is composed of a sequence of two-phase commit processes of its actions.

Compared with transaction-level two-phase commit mechanism, action-level is more flexible on resource usages, more adaptive to changing environment, and more reliable considering possible exceptions in commit phase. However, it involves more computation and communication overheads because several rounds of resource reservations are required (one round for each action), while transaction-level requires only one round for a whole transaction. Action-level two-phase commit is preferred for CPSs with changeable operating environment and irregular operations, stringent requirements on absence of exceptions, or preference of real-time reactions. Transaction-level is more suitable for CPSs that require less running overheads and whose operations are more stable and predictable based on laws of physics.

In this paper, we focus only on how to apply transaction-level two-phase commit to process transactions.

5.3 Pre-Commit Phase and Conflicts

When potential conflicts are found between pre-Write sets of different transactions, measures should be taken to resolve these conflicts. In this section, we first consider how to resolve conflicts between two transactions, and later we would discuss how to resolve conflicts among multiple (more than two) transactions.

Assume that T_i is currently being scheduled (in *Scheduling* state) by CPS entity E_i , and T_j is a transaction of another CPS entity E_j . Potential conflicts between two transactions T_i and T_j are identified by comparing their pre-Write sets. If

$$\text{preUResr}_{T_i, s_{si}} \bigcap_{\text{overlap}} \text{preUResr}_{T_j, s_{sj}} \neq \emptyset,$$

then T_i and T_j have potential conflicts.

5.3.1 Conflict Types

Given states of two conflicting transactions, two conflict types are defined:

- **PP Conflict.** *PP Conflict* occurs between two transactions in their pre-commit phases (in either *Triggered* or *Scheduling* state). Since conflicts are checked only when one transaction sends out reservation requests (in *Scheduling* state), we consider only one transaction in *Scheduling* state and another in *Triggered* or *Scheduling* state.
- **PC Conflict.** *PC Conflict* occurs between one transaction in the pre-commit phase and another in the commit phase (in either *PreCommitted* or *Executing* state). For the same reason as PP Conflict, we consider only one transaction in *Scheduling* state and another in *PreCommitted* or *Executing* state.
- **CC Conflict.** *CC Conflict* occurs between two transactions in their commit phase (*Executing* state). Such a conflict indicates a conflict in realtime is already occurring and it is too late to take precautions to avoid them. Thus, we will ignore CC Conflict when discussing conflict resolution strategies.

5.3.2 Transaction Precedence

When potential conflicts are found between T_i and T_j , a weighing process is performed (by either E_i or E_j) to determine their precedence based on three elements: transaction priority, transaction execution time, and transaction state. After this, depending on conflicts types, different strategies are used to resolve potential conflicts.

Transaction priority is computed given the context where a transaction is triggered and what task a transaction is designed to complete. The context includes the start system state and the trigger event. For example, transactions of cars in left lanes are usually given higher priorities than those of cars in right lanes, and transactions triggered for handling abnormal situations have higher priority than regular transactions. Specially, we assign a particular class of priorities to transactions triggered for handling emergency situations. Such transactions are given a priority equal to or higher than $\mathbf{pri_{PE}}$.

Transaction execution time is the expected start and end execution time of the transaction. Here, we assume that a synchronized clock is maintained in a CPS Network to provide time-related services to CPS entities.

Transaction state indicates the state of a transaction in two-phase commit processing (Figure 5.1). Since T_i is currently under scheduling, it is in *Scheduling* state. But T_j can be in one of four possible states: *Triggered*, *Scheduling*, *PreCommitted*, and *Executing*. Thus, two types of conflicts exist: PP Conflict and PC Conflict.

5.3.3 PP Conflict and Resolution Strategy

Assume that T_i is in *Scheduling* state and T_j is in *Triggered* or *Scheduling* state, then conflicts between them are of type PC Conflict. Assume their priorities are pri_{T_i} and pri_{T_j} respectively, and their execution times are $[ts_{T_i}, te_{T_i})$ and $[ts_{T_j}, te_{T_j})$. The precedence between T_i and T_j is decided as below (Figure 5.2).

- If T_j is in *Triggered* state, then T_i wins. This is because resource reservation for T_j has not started.

- If T_j is in *Scheduling* state, then
 - If $pri_{T_i} \neq pri_{T_j}$, then the one with the higher priority wins.
 - If $pri_{T_i} = pri_{T_j}$, then we compare ts_{T_i} and ts_{T_j} , and the one with the earlier start time wins. If start times are equal, then we use te_{T_i} and te_{T_j} to decide.
 - If both priority and execution time are the same, then we use other information to decide the winner such as location (e.g., left-lane or front cars are preferred) (NOT shown in Figure 5.2).

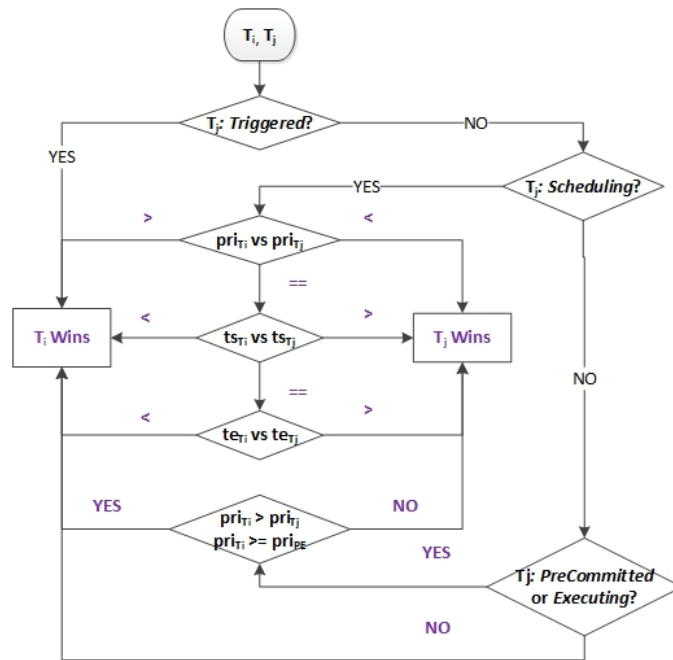


Figure 5.2: Weighing Process Between T_i and T_j

There are three strategies to resolve a potential PP Conflict: Win-Lose, Win-Win, and Enhanced Win-Win.

In the **Win-Lose** strategy, the transaction with the higher precedence can reserve desired access times for specified resources, but the other can't and is aborted. Thus, only one of the two conflicting transactions survives.

The **Win-Win** strategy tries to keep both transactions alive. Similar as the Win-Lose strategy, the winning transaction can reserve its pre-Write set. However, for the losing one,

it is adjusted rather than aborted. The transaction adjustment process (defined later) allows *ComS* to adjust a conflicting transaction to produce a different pre-Write set that has no conflicts with the other transaction. If such an adjustment works, the transaction conflicts are resolved and both transactions (one as adjusted) get the requested resources reserved. If not, then the losing transaction is aborted. Thus, Win-Win strategy saves at least one out of two conflicting transactions.

The **Enhanced Win-Win** strategy improves Win-Win strategy by requiring both conflicting transactions, rather than only the losing one, to be adjusted until no conflict is found between them. In practice, the Win-Win strategy is applied first. If it fails, then we try Enhanced Win-Win. If Enhanced Win-Win doesn't work either, then the winning transaction is kept, and the losing one is aborted.

5.3.4 PC Conflict and Resolution Strategy

If T_i in *Scheduling* state and T_j is in *PreCommitted* or *Executing* state then the conflict between them is of type PC Conflict. The precedence between T_i and T_j is determined as follows (Figure 5.2).

- If $pri_{T_i} > pri_{T_j}$ and $pri_{T_i} \geq \mathbf{pri_{PE}}$, then T_i wins.
- Otherwise, T_j wins.

For a PC Conflict, a new conflict resolving strategy is introduced: **transaction preemption**. In transaction preemption, the execution of one transaction is canceled to give way to another transaction. The above weighing process indicates that for a transaction in state *PreCommitted* or *Executing*, it will be preempted only when the conflicting transaction has a higher priority and the priority reaches emergency level ($\mathbf{pri_{PE}}$).

Such a strategy is based on the fact that it takes effort for a CPS entity to schedule (or pre-commit) a transaction, e.g., negotiation with other CPS entities and possible transaction adjustments. Moreover, the preempted transaction may already have partially

executed. So, the preemption of a *PreCommitted* or *Executing* transaction is considered only in safety-critical situations. \mathbf{pri}_{PE} is the threshold of transaction priority to decide whether transaction preemption can be applied.

Based on the weighing process, PC conflict is resolved in following way:

- If T_i loses in precedence weighing, then T_i is either aborted (Win-Lose) or adjusted (Win-Win) to avoid conflicts.
- If T_i wins, then $\mathbf{preUResr}_{T_i, S_{si}}$ can be reserved, and transaction preemption is performed on T_j :
 - If T_j is in *PreCommitted* state, then it is aborted. Its resource reservation is revoked and reserved resources are released. T_j 's state becomes *Aborted*.
 - If T_j is in *Executing* state, then its execution is interrupted, and corresponding resource reservation is revoked. T_j becomes *Committed* and *Incomplete*.

5.3.5 Dynamic Transaction Adjustment

Transaction adjustment was introduced earlier as a way to adjust a conflicting transaction. Transaction adjustment replaces a conflicting transaction with an alternative one of the same type, but with a different pre-Write set to achieve the same goal.

From our definitions of a resource usage set, we know that changing action attribute values or start system state will change the resource usage set of an action. Two types of adjustments can be applied to a transaction: action adjustment and sequence adjustment.

- **Action adjustment** adjust actions of a transaction by changing their attribute values. It replaces existing actions with others of the same types but with different attribute values.
- **Sequence adjustment** adjusts start system states of a transaction or its actions. The adjustment is fulfilled by inserting *temporary* actions into the action sequence,

e.g., *IncreaseSpeed* in Figure 4.1. Temporary actions will change system states of a CPS and thus change the start system state of a transaction or an action.

5.3.6 Pre-Commit Phase: Conflicts in Multiple Transactions

In the previous sections, we discussed how to resolve conflicts between two transactions. However, to reserve $\mathbf{preUResr}_{T_i, S_{si}}$, a CPS entity E_i has to check conflicts with all other CPS entities (rather than just E_j) in a CPS Network. It is very likely that T_i has potential conflicts with more than one transaction from different CPS entities. In such a case, conflict resolution becomes more complicated because T_i has to consider and resolve potential conflicts with multiple transactions at the same time.

For each transaction conflicting with T_i , a weighing process is performed to decide their precedence. Then

- If T_i wins in all weighing processes (either PP or PC Conflict), then $\mathbf{preUResr}_{T_i, S_{si}}$ can be reserved. For each conflicting transaction T_j ,
 - if it has PP Conflict with T_i , then it is aborted or adjusted.
 - if it has PC Conflict with T_j , then it is preempted and its resource reservation is revoked.
- Otherwise, reservation can not be made, and T_i is aborted or adjusted.

5.4 Pre-Commit Phase: Transaction Processing Algorithm

We have so far described how transaction conflicts between transactions are resolved in the pre-commit phase. In this and next section, we give specific algorithms of how a transaction is processed in the pre-commit phase and how conflict resolution strategies are applied. We assume that the communication network in CPS Network is reliable and no message error would occur. Depending on whether there is an environmental resource server in a

CPS Network, there are two different transaction processing mechanisms: *centralized* and *distributed*.

5.4.1 State of a Reservation

Resource reservation for a transaction starts when a CPS entity sends out a reservation request. The reservation goes through different states in the reservation process (Figure 5.3).

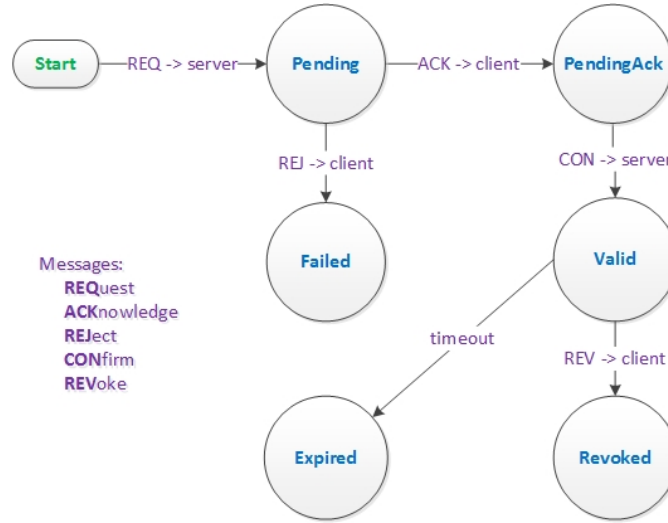


Figure 5.3: State Machine of Resource Reservation

- A resource reservation becomes *Pending* after the CPS entity has sent out a corresponding request (REQ), but before the request is granted by the resource server.
- If the corresponding request has been acknowledged (ACK) by the server, then a pending reservation becomes *PendingAck*. When the client confirms the reservation (CON), the reservation becomes *Valid*.
- If a reservation request is rejected by the server (REJ), then a pending reservation becomes *Failed*.
- A valid reservation becomes *Revoked* if it is revoked (REV) by the server or withdrawn by the CPS entity itself before it become *Expired*. A valid reservation expires when the last usage time for one of those reserved resources expires.

5.4.2 Resource Client and Server in Centralized Environment

In centralized transaction processing, resource usages of all CPS entities in a CPS Network are managed by a dedicated *resource server*, and all CPS entities (E_i , $1 \leq i \leq n$) in the network are *resource clients*. All resource reservations are handled by the server, and the server has the right to grant or reject reservation requests.

Each time a CPS entity E_i starts scheduling a transaction T_i some time before T_i 's expected start execution time, a resource reservation request REQ_i containing $\mathbf{preUResr}_{T_i, S_{si}}$ (S_{si} is the expected start system state of T_i) is sent to the server.

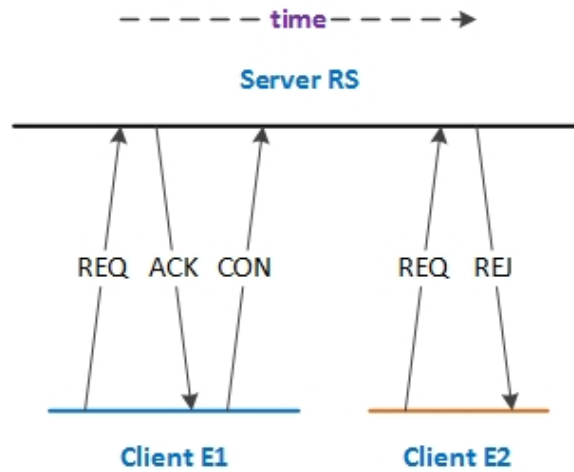


Figure 5.4: Messages Between Client and Server

Resource Server

When the resource server receives an incoming request message REQ_i for T_i from E_i , it performs following procedures to process the request.

1. Find transactions whose execution time period overlaps with T_i 's. Perform $\bigcap_{overlap}$ operation to detect potential conflicts. There are three types of transactions:
 - Transactions with *Valid* reservations: These transactions are either in *PreCommitted* or *Executing* state, and PC Conflict applies.

- Transactions with *PendingAck* reservations: These transactions are in *Scheduling* state, and PP Conflict applies.
 - Transaction with *Pending* reservations” These transactions are also in *Scheduling* state and PP Conflict applies. However, for this type of transactions, the First-Come-First-Server mechanism is taken, i.e., only transactions whose requests arrive before REQ_i are checked.
2. If a conflict exists, then it performs a weighing process between T_i and each of the conflicting transactions. The weighing process contains checking transaction priorities, execution times, and conflict types, as described in previous section. When weighing process is done, precedence between transactions is determined.
- If T_i 's reservation can be made (T_i wins all), and if the reservation requires revoking some *Valid* or *PendingAck* reservations, a REV (Revoke) message is sent to each corresponding CPS entity to revoke their reservations. When revoking is done, send an ACK (Acknowledge) message to E_i indicating that **preUResr** _{T_i, S_{si}} can be reserved.
 - Otherwise, send a REJ (Reject) message to E_i indicating unsuccessful reservation.
3. If no conflict has been found, send an ACK message to E_i indicating successful reservation.

Resource Client

In each reservation process, after sending the request, a resource client waits for response messages from the server.

- If an ACK message is received, then it sends a CON (Confirm) message back to confirm the reservation (Client $E1$ in Figure 5.4). T_i now becomes *PreCommitted* and can be scheduled to execute.

- If a REJ message is received, then it aborts or adjusts T_i (Client E_2 in Figure 5.4).

For each client that receives a REV message, its reservation is revoked by the server. In this case, operations such as abortion or adjustment of corresponding transactions need to be taken.

5.4.3 Resource Client and Server in Distributed Environment

Without a dedicated management entity, resource reservation for a transaction is then a negotiation process with other CPS entities in a CPS Network. Only when all other CPS entities confirm that they don't have transactions in conflict with a transaction, it can be scheduled. In this scenario, each CPS entity plays both resource server and client roles. For the server role, a CPS entity receives resource reservation requests from other CPS entities and check whether they can reserve specified resources. For the client role, when a CPS entity is scheduling one of its transactions, a reservation request is to all other CPS entities.

Resource Server

When a CPS entity E_s receives an incoming request message REQ_c from a CPS entity E_c to reserve resources for its transaction T_c , it plays the server role and performs a sequence of operations to decide whether $\mathbf{preUResr}_{T_c, S_{sc}}$ can be reserved. The process is similar to that of the centralized resource server, except that E_s now checks its transactions only against T_c . The process is shown in Figure 5.5.

E_s first determines whether any of its transactions has *overlapped* execution time with that of T_c . Let $\mathbf{T}_{\mathbf{over}}$ be the set that contains all overlapped transactions of E_s . Following evaluation is performed.

1. If $\mathbf{T}_{\mathbf{over}}$ is empty (Condition C_1 in Figure 5.5), no conflicts would be found. Send an ACK message to E_c to indicate that reservation can be made in E_s .
2. Otherwise, for each transaction in $\mathbf{T}_{\mathbf{over}}$, compare its pre-Write set with that of T_c . Assume that $\mathbf{T}_{\mathbf{conf}}$ is a set of conflicting transactions against T_c . The evaluation

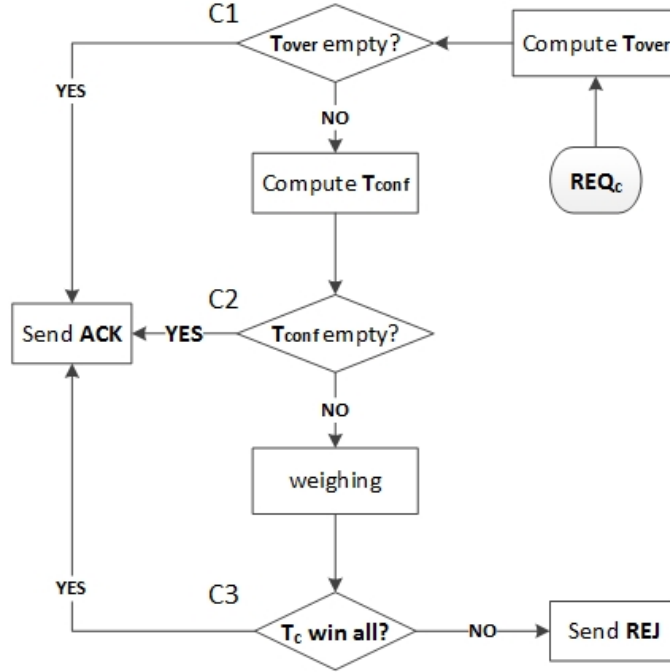


Figure 5.5: Algorithm for Resource Server: Receiver of REQ_c

continues.

- If \mathbf{T}_{conf} is empty (Condition $C2$ in Figure 5.5), no conflicts are found. Send an ACK message back to E_c .
- Otherwise, for each $T_s \in \mathbf{T}_{\text{conf}}$, perform weighing process and decide whether T_c can win (Condition $C3$ in Figure 5.5).
 - If T_c wins in all weighing processes against transactions in \mathbf{T}_{conf} , an ACK response is sent to E_c .
 - If not, a REJ response is sent.

If E_s sends a REJ message to E_c , the evaluation process for reservation of $\mathbf{preUResr}_{T_c, S_{sc}}$ completes. If E_s sends an ACK message to E_c previously, then E_s waits for response from E_c .

- If a CAN (Cancel) message is received from E_c , reservation of $\mathbf{preUResr}_{T_c, S_{sc}}$ for T_c fails.

- If a CON (Confirm) message is received from E_c , reservation of $\mathbf{preUResr}_{T_c, S_{sc}}$ is made. E_s keeps a local record of $\mathbf{preUResr}_{T_c, S_{sc}}$, and then following operations are taken for each transaction T_s in $\mathbf{T}_{\mathbf{conf}}$.
 1. If T_s has PP Conflict with T_c , it is either aborted or adjusted.
 2. If T_s has PC Conflict with T_c , it is preempted and its resource reservation is revoked by sending a REV message to all other CPS entities.

Resource Client

Figure 5.6 shows the process followed by CPS entity E_c to reserve resources for its transaction T_c . The role E_c plays here is client, and other CPS entities are servers. Different from centralized resource client, E_c now needs to consider responses from all other CPS entities).

The scheduling (or reservation) process performs two levels of conflict checking: internal and external check. **Internal** conflict check is performed locally in E_c to see whether conflicts exist between T_c and existing resource reservations recorded locally by E_c (when E_c plays the server role). **External** conflict check is performed in other CPS entities who play the server role after a source reservation request is sent out from T_c .

In internal conflict check, existing reservations from other CPS entities are in one of *Pending*, *PendingAck* and *Valid* states. Only those that have overlapped execution time with $[ts_{T_c}, te_{T_c})$ are examined (indicated by $\mathbf{VT}_{\mathbf{over}}$). The process is similar to what a centralized resource server does and we won't give details here.

- If conflicts (PP or PC Conflict) exist, and T_c wins in all weighing processes (Condition $C1$ and $C2$ in Figure 5.6), continues to external check.
- Otherwise, reservation for T_c would not succeed even if a request is sent. So there is no need to send out a reservation request, and T_c is either adjusted or aborted.

If no conflicts are found in internal check or T_c has the highest priority, external check can be carried out and a reservation request message \mathbf{REQ}_c is sent. E_c waits for responses

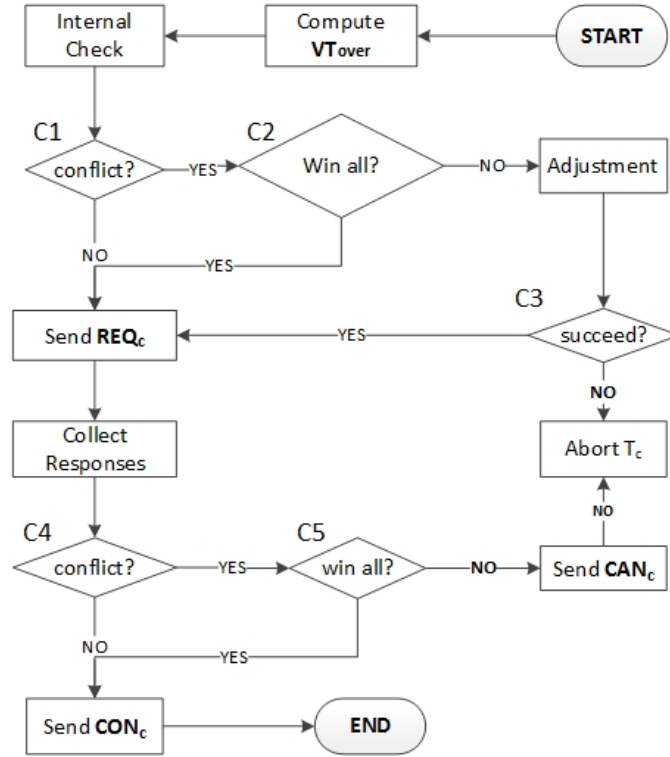


Figure 5.6: Algorithm for Resource Client: Sender of REQ_c

from all other entities in the network. Conditions $C4$ and $C5$ in Figure 5.6 evaluate whether a reservation can be made:

1. If no conflicts are found, the reservation can be made.
2. If conflicts are found, but T_c wins in all weighing processes (both PP and PC conflicts), the reservation can be made.
3. If conflicts are found, but T_c doesn't win in all weighing processes, the reservation can't be made.

In above, first two cases correspond to ACK messages from all other CPS entities. The third corresponds to one or more REJ messages from other CPS entities.

If the reservation can be made, a CON message is sent to confirm the reservation of T_c to all other CPS entities. This confirmation message would trigger a recipient entity who

has sent a response back previously to abort or adjust transactions if it has transactions in conflict with T_c .

If the reservation can't be made, a CAN message is sent to withdraw the reservation for T_c , and then T_c is either aborted or adjusted.

5.5 Commit Phase and Exceptions

When a transaction is scheduled, it becomes pre-committed and waits for its turn to be executed by *PhyS*. As mentioned before, the execution process transforms each action of a transaction into low-level control commands upon physical devices, which perform physical operations and create desired physical effects. As for how control commands manipulate physical devices, it is out of the scope of our research, and we assume that such manipulation always works in expected ways. We will, however, focus on how execution is monitored.

The execution of a transaction (T_i) is supposed to follow what has been predicted, i.e., its Write set $\mathbf{UResr}_{T_i, S'_{si}}$ should match its pre-Write set $\mathbf{preURResr}_{T_i, S_{si}}$. This requires that the realtime start system state, S'_{si} , equals to the predicted start system state S_{si} . If the start system state changes, then realtime behavior would be different from what is expected.

5.5.1 Exception

An **exception** occurs when realtime resource usage is different from the predicted. Two types of exceptions can happen when executing a transaction T_i .

Internal Exception

An exception occurs when any environmental resource is not used as predicted by T_i . Assume that execution of T_i uses resource res' in time period $[t'_s, t'_e]$. We have $(res', [t'_s, t'_e]) \in \mathbf{URResr}_{T_i, S'_{si}}$. If $(res', [t'_s, t'_e]) \notin \mathbf{preURResr}_{T_i, S_{si}}$, then an exception occurs. It has two cases:

1. $\forall (res, [t_s, t_e]) \in \mathbf{preURResr}_{T_i, S_{si}}, res \neq res'$.

2. $\forall(res, [t_s, t_e]) \in \mathbf{preUResr}_{T_i, S_{si}}, [t_s, t_e] \neq [t'_s, t'_e]$ for any $res = res'$.

The first case indicates using unspecified resources and the second indicates using specified resources in unreserved time periods. Both cases cause discrepancy between pre-Write and Write sets.

External Exception

An exception can also occur when there existing another transaction T_j from another CPS entity E_j uses resources in time periods reserved by T_i (assume that T_i will be executed as predicted), i.e.,

$$\mathbf{preUResr}_{T_i, S_{si}} \bigcap_{\text{overlap}} \mathbf{UResr}_{T_j, S_{sj}} \neq \emptyset.$$

Although E_i has already made a reservation for T_i and other CPS entities are not allowed to use the same resources in time periods reserved by T_i , the case above can still occur if E_j malfunctions.

5.5.2 Exception Detection and Handling

To detect potential exceptions, execution of T_i is monitored. The main approach is to do a resource availability check before executing T_i . The checking process is carried out some time before the start execution time of T_i , so that enough time is left for exception handling in case of potential exceptions. The checking process includes internal check and external check. Internal check detects potential internal exceptions, while external check detects potential external exceptions.

Internal Check makes sure the realtime start system state (S'_{si}) of T_i will be the same as the predicted one (S_{si}). However, since internal check is performed (a little time) before the execution of T_i , the realtime state here is not actually “realtime”, but a more accurate and reliable prediction of the start system state than S_{si} . If S'_{si} is different from S_{si} , potential exception exists.

The approach to handle potential internal exception includes two steps.

- The first step is to adjust T_i to see whether the adjusted transaction can be executed using $\mathbf{preUResr}_{T_i, S_{si}}$ under the new start system state S'_{si} without causing conflicts.
- If the first step doesn't work, then reserve new resources so that these newly reserved resources, together with resources in $\mathbf{preUResr}_{T_i, S_{si}}$ (still a *Valid* reservation), can meet resource requirements when executing T_i under start state S'_{si} . When reserving new resources for T_i , its transaction priority is promoted to give it a better chance to reserve requested resources.

If above two steps are not working, execution of T_i is canceled and T_i becomes *Committed* and *Incomplete*.

External Check detects makes sure usage time periods for specified resources reserved by T_i is not used by other transactions. To identify potential conflicting usages, a checking message is sent to the resource server to secure reservation of T_i .

For potential external exception, considering that it is caused by several kinds of events, such as malfunction of scheduling algorithm, communication malfunction, or involvement in emergency situations of another CPS entity, the handling approach first figures out the cause and takes corresponding measures afterwards. Assume that the conflicting transaction that causes the external exceptions is T_j of CPS entity E_j . The procedure is described below.

- If $pri_{T_j} \geq pri_{PE}$ and $pri_{T_j} > pri_{T_i}$, then T_j is triggered to handle emergency situations. Transaction preemption is applied. T_i is preempted by T_j .
- Otherwise, a negotiation message is sent to notify E_j that an exception is caused by T_j and T_j should be canceled. a timer is set.
 - If no response is received before the timer expires, E_j is assume to malfunction. Further negotiation with E_j will not help. T_i is preempted by T_j .
 - If a positive response indicating cancellation of T_j is received within the timer, T_i is kept and execution continues.

- If a negative response is received within the timer, T_i is preempted by T_j .

When T_i is preempted, Execution of T_i is canceled, and reservation of $\mathbf{preUResr}_{T_i, S_{si}}$ is revoked. T_i becomes *Committed* and *Incomplete*.

5.5.3 Runtime Exception

Potential internal and external exceptions are detected and handled some time before execution of a transaction starts. However, if an exception still occurs when executing a transaction, then bad consequences may be caused. This type of exception happens in realtime, and is detected only after it occurs. We call it **Runtime Exception**.

The effect of a runtime exception depends on whether conflicting usage of the same resources exist. If conflicting usages exist, a runtime exception causes undesired consequences, such as collision of cars, and it is too late to avoid them. However, actions can be taken to prevent chain effects such as emergency warning message propagation [57]. If no conflicting usages exist, measures should be taken to prevent potential transaction conflicts for remaining transactions waiting to be executed.

5.6 Summary

We have presented the two-phase commit transaction processing algorithm in this chapter.

In the pre-commit phase, a transaction is scheduled by reserving required resources in the resource server. In a centralized resource management environment, the resource server is a central entity that manages all resource usages, and, in a distributed environment, the resource server indicates all other CPS entities in the CPS Network. The CPS entity sending out the reservation request is the client. If potential conflicts are detected, they are resolved by the CPS entity to prevent real-time transaction conflicts. Two types of conflicts may occur in the pre-commit phase: PP and PC Conflict. Three resolving strategies are proposed to resolve PP Conflict: Win-Lose, Win-Win, and Enhanced Win-Win. For PC Conflict, an additional strategy is used: Transaction Preemption. Two transaction processing algorithms

for the centralized and distributed environment are proposed. In each algorithm, we show how transaction scheduling is carried out for both the resource client and server.

While it is desirable that the commit phase is performed as what has been predicted in the pre-commit phase, exceptions still happen. An exception is caused either internally or externally. To prevent real-time transaction conflicts, exception detection and handling are required. The detection process consists of an internal check and an external check, which detect internal and external exceptions respectively. When an internal exception is detected, a transaction is either aborted or adjusted. When an external exception is detected, a negotiation effort is first made to keep reserved resources, and, if it fails, the transaction is aborted or adjusted.

In next chapter, we show how a simulation platform is implemented and how the two-phase commit algorithm is used by different components of a CPS entity to process transactions.

Chapter 6

Simulation Platform Implementation: CPSNET

6.1 Introduction

In this chapter, we show how the simulation platform (CPSNET) for our transaction model is implemented using the Java programming language [58]. The implementation is based on the definitions and algorithms of the transaction model proposed in Chapter 4 and 5 respectively.

The simulation platform simulates a CPS Network, where different CPS entities coordinate resource usages of their transactions through a Communication Network. Depending on the resource management mechanism, a Resource Server may exist in the CPS Network to manage resource usages of all CPS entities. In a CPS entity, four main components are implemented to simulate the computational subsystem *ComS* and the physical subsystem *PhyS*: Transaction Generator, Transaction Scheduler, Transaction Executor, and Resource Manager. These components play different roles in processing a transaction from triggering, scheduling, to executing a transaction. However, they are not independent, and one component may rely on another to fulfill its function, such as that Transaction Scheduler component relies on Resource Manager component to query resource usage information. Besides, there are additional components which play specific roles with respect to processing a transaction, such as Conflict Resolver component resolving transaction conflicts in

the pre-commit phase and Exception Handler component handling exceptions when executing a transaction in the commit phase. Details of the design and implementation of these components will be presented in this chapter.

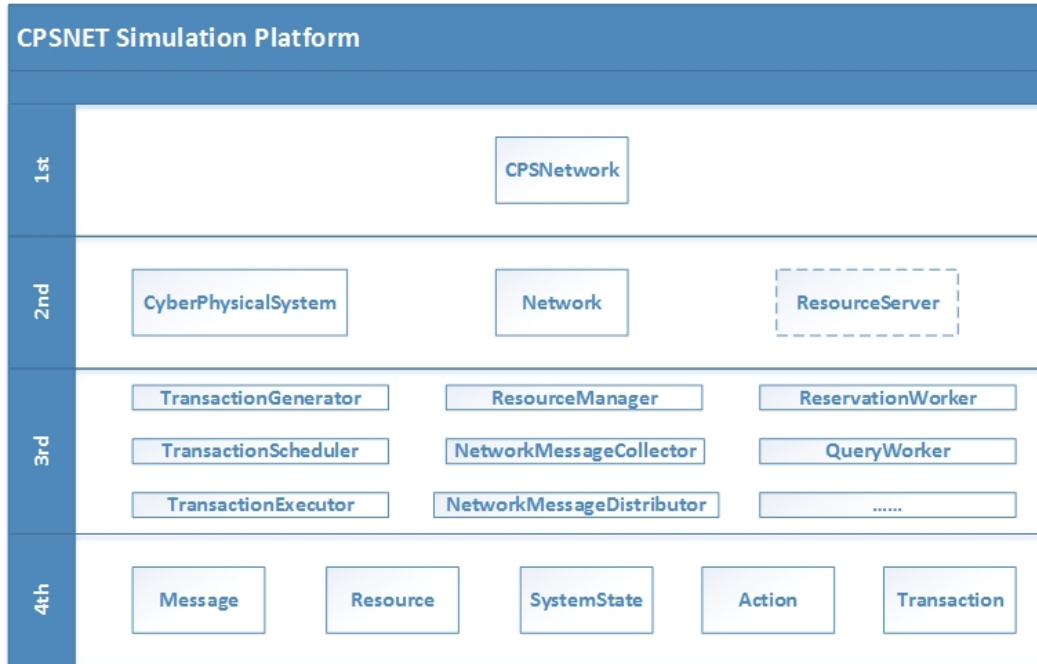


Figure 6.1: CPSNET Simulation Platform

The simulation platform is divided into four levels (Figure 6.1).

1. The 1st level is *CPSNetwork*, which represents the operating environment of a Cyber-Physical System (Section 4.3). *CPSNetwork* organizes different entities in the 2nd level into an interconnected network and serves as the simulation entry point. It carries out a time-based simulation and updates each CPS entity's system state based on their current transactions at fixed time interval.
2. The 2nd level contains different types of entities, i.e., a *Network*, an optional *ResourceServer*, and a group of *CyberPhysicalSystems*. *Network* is the Communication Network that supports communication among CPS entities. *ResourceServer* represents the Resource Server in a CPS Network that manages resource usages. *CyberPhysical-*

System represents a Cyber-Physical System that performs physical operations using environmental resources to achieve a certain goal.

3. Entities in the 2nd level are composed of components in the 3rd level. For example, a *CyberPhysicalSystem* consists of four main components: *TransactionGenerator*, *TransactionScheduler*, *TransactionExecutor*, and *ResourceManager*. As mentioned above, inter-dependency may exist between components, such as *TransactionScheduler* depending on *ResourceManager* to perform resource reservations and *ConflictResolver* to resolve transaction conflicts.
4. Classes in the 4th level provide data or operation abstractions that support entities in the above three levels, such as *Message*, *Resource*, *ResourceUsage*, *SystemState*, *Action*, and *Transaction*.

As the simulation entry point, when *CPSNetwork* is started, it performs a sequence of procedures to initialize and start the simulation, as shown in Figure 6.2. It first reads in configuration files, and configures the simulation environment. Then it initializes all entities in the 2nd level, i.e., *Network*, *ResourceServer*, and *CyberPhysicalSystems*. When initialization is completed, the simulation is started by starting all entities. All these entities operate independently, and interact with each other through messages. The simulation process is controlled by a global clock that advances the time periodically. When the predefined simulation time is reached, the simulation is terminated. A set of statistical information is collected by *CPSNetwork*. These information are from entities in the 2nd level, which in turn obtain the information from their components in the 3rd level.

In the following sections, we present our design and implementation of the simulation platform in a bottom-up manner. We first introduce the implementation of *Resource* and *ResourceUsage*, then present the implementation of *Action* and *Transaction*. After that, we show how different entities in the 2nd level and their components in the 3rd level are implemented. At last, we give more details of *CPSNetwork* and show how the simulation

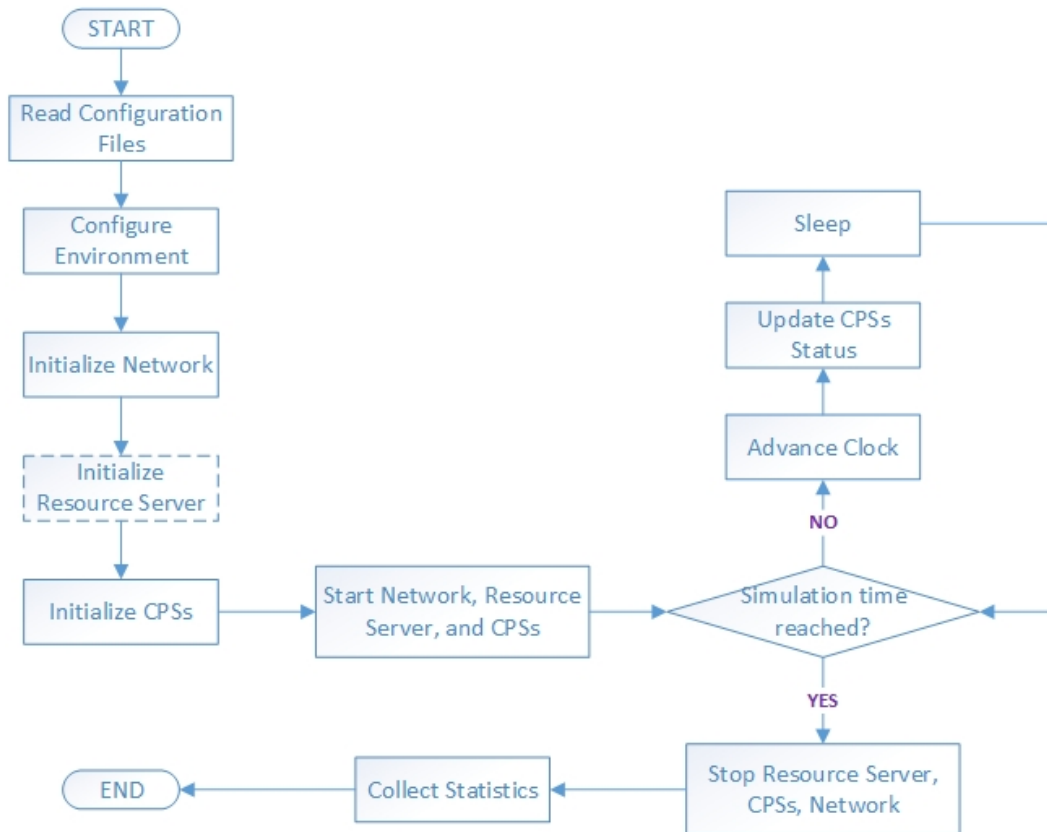


Figure 6.2: Simulation Flow

flow in Figure 6.2 is executed. We continue using the autonomous car example in the implementation.

6.2 Resource and Resource Usage

We define three classes related to environmental resources: *Resource*, *ResourceUsage*, and *ResourceUsageOperations*. Their relationship is shown in Figure 6.3.

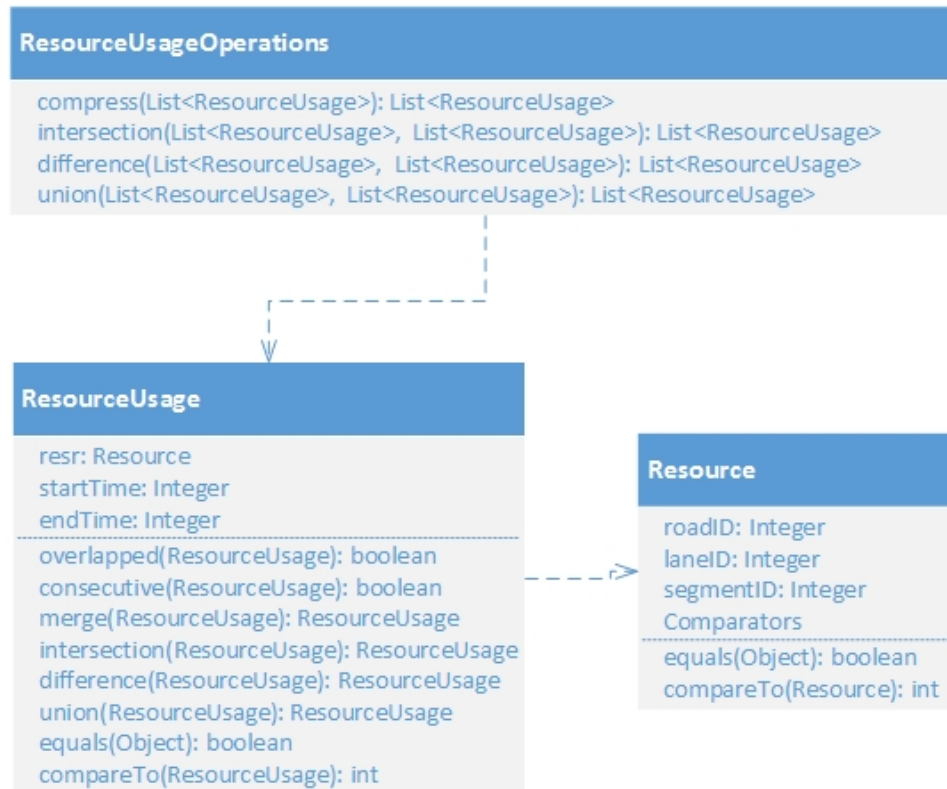


Figure 6.3: Resource, ResourceUsage, and ResourceUsageOperations

6.2.1 Resource

The *Resource* class provides an abstraction of a single environmental resource. In the car example, assuming that a road is represented by a grid map, a *Resource* object represents a grid and contains three fields (Figure 6.3):

- *roadID* is the identification of a road.

- *laneID* is the identification of a lane in a road.
- *segmentID* is the identification of a segment in a lane.

Besides implementing the *Comparable* interface, two *Comparators* are defined in the *Resource* class to support two types of ordering: `SEG_LANE_ROAD_ORDER` and `ROAD_LANE_SEG_ORDER`. In `SEG_LANE_ROAD_ORDER`, *Resource* objects are first sorted by *segmentID*, then by *laneID*, and lastly by *roadID*, while `ROAD_LANE_SEG_ORDER` applies an opposite order. `SEG_LANE_ROAD_ORDER` is considered as the natural order of *Resource* objects.

6.2.2 Resource Usage

ResourceUsage represents the usage of a resource. Each *ResourceUsage* object contains a reference to a *Resource* object, and two additional fields indicating the time period when a resource is being used: *startTime* and *endTime*. Here, we use $(res, [startTime, endTime])$ to represent a *ResourceUsage* object.

Several methods are provided in the *ResourceUsage* class. The following are examples of these methods:

- *overlapped*: checks whether two resource usages are overlapping. For example, $(res_0, [0, 5])$ and $(res_0, [3, 8])$ are overlapping, but $(res_0, [0, 5])$ and $(res_0, [5, 8])$ are not. If two resource usages are of different resources, they do not overlap, e.g., $(res_0, [0, 5])$ and $(res_1, [3, 8])$.
- *consecutive*: checks whether two resource usages are consecutive, e.g., $(res_0, [0, 5])$ and $(res_0, [5, 8])$ are consecutive, and $(res_0, [0, 5])$ and $(res_0, [3, 8])$ are consecutive.
- *merge*: merges two consecutive resource usages. $(res_0, [0, 5])$ merged with $(res_0, [3, 8])$ gives $(res_0, [0, 8])$, and $(res_0, [0, 5])$ merged with $(res_0, [5, 8])$ gives $(res_0, [0, 8])$. However, $(res_0, [0, 5])$ merged with $(res_1, [3, 8])$ gives *null*.

- *intersection*: finds the intersected usage between two resource usages, e.g., the intersection of $(res_0, [0, 5))$ and $(res_0, [3, 8))$ is $(res_0, [3, 5))$.
- *difference*: calculates the difference of one resource usage with respect to another. For example, the difference of $(res_0, [0, 5))$ from $(res_0, [2, 3))$ is $(res_0, [0, 2))$ and $(res_0, [3, 5))$, and $(res_0, [0, 5))$ from $(res_1, [2, 3))$ is $(res_0, [0, 5))$.
- *union*: returns the union of two resource usages. For example, union of $(res_0, [0, 5))$ and $(res_0, [5, 8))$ gives $\{(res_0, [0, 8))\}$, and union of $(res_0, [0, 5))$ and $(res_1, [2, 3))$ gives $\{(res_0, [0, 5)), (res_1, [2, 3))\}$. The difference between *merge* and *union* is that if two resource usages are of different resources, or they are not consecutive or overlapping, *null* is returned for method *merge*, whereas a set containing both resource usages is returned for *union*.

Two *Comparators* are defined to support ordering by resource (RESOURCE_ORDER) and ordering by usage time (TIME_ORDER), and ordering by usage time is the natural order of *ResourceUsage* objects.

6.2.3 Resource Usage Operations

The *ResourceUsageOperations* class provides a set of static methods that operate on lists of *ResourceUsage* objects.

- Method *compress* merges resource usages of the same resource that have consecutive or overlapped execution time periods in a list.
- Method *intersection* returns intersected resource usages between two lists of resource usages.
- Method *difference* returns different resource usages in the first list with respect to those in the second list.
- Method *union* returns the union of resource usages in two lists.

ResourceUsage objects in lists returned by these methods are sorted by TIME_ORDER.

6.3 Action and Transaction

Actions and transactions provide different levels of abstractions for physical operations of a CPS entity. In the implementation, *Action* and *Transaction* are both defined as abstract classes, from which concrete sub-classes are derived. Each sub-class defines a specific type of action or transaction. Properties such as start and end system states, action sequence, and resource usage set are defined as fields in *Action* and *Transaction*.

6.3.1 System State

As we have described in Chapter 4, a system state represents the status of a CPS entity in a CPS Network. It serves as the operating context for actions and transactions. For the car example, we define a *SystemState* class that contains speed, acceleration and location of a car. The location is given by a *Resource* object, which corresponds to a specific grid in a grid map.

6.3.2 Action

The abstract class *Action* defines common fields and methods shared by different types of actions, and each type of action is defined by a class which inherits fields and methods from *Action*. In our car example, seven sub-classes are defined, as shown in Figure 6.4:

Main fields defined in *Action* include:

- *actionID* is a unique id that identifies an action.
- *actionType* is the type of an action. Seven action types are defined for the car example: *ConstantSpeedAction*, *IncreaseSpeedAction*, *DecreaseSpeedAction*, *TurnLeftAction*, *TurnRightAction*, *TurnOnSignalAction*, and *TurnOffSignalAction*.
- *priority* is the priority of an action, which is used in action-level two-phase commit processing (Section 5.2.4). Four levels of priorities are defined: 1 ~ 4. These levels

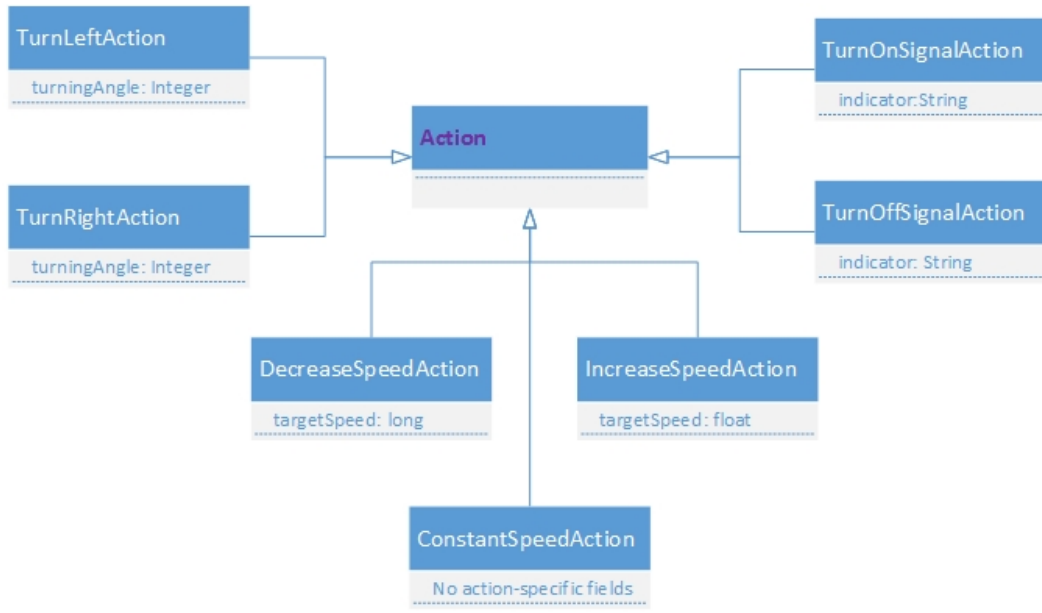


Figure 6.4: Action

correspond to the purpose of an action. Level 4 indicates a normal action to be scheduled for execution, level 3 is a conflict-resolving or cooperative action, level 2 is an exception handling action, and level 1 is an emergency-handling action.

- *startState* and *endState* are the start and end system state of an action respectively. They are of type *SystemState*.
- *startTime* and *endTime* are the start and end execution time of an action respectively. Timing is supported by a simulated global clock (discussed later).
- *writeSet* is the resource usage set of an action. It has type *List<ResourceUsage>*. The *writeSet* may represent the pre-Write or Write set depending on the transaction processing phase the action is undergoing. If it is transaction scheduling, then *writeSet* indicates *pre-Write* set; if it is transaction executing, then *writeSet* indicates *Write* set.
- *Comparators* are different comparator classes supporting ordering actions by *actionID*, *actionType*, *priority*, and the execution time (*startTime* and *endTime*).

The main methods defined in *Action* include:

- Constructors provide several ways to initializing an action.
- Getter and setter methods allows getting and setting field values.
- WriteSet operation methods that operate on *writeSet*: *intersection*, *difference*, and *union*. These operations are based on methods defined in classes *ActionOperations* and *ResourceUsageOperations*.
- Abstract method *updateFields()* is a method called by constructors to help initialize an action instance. This method is action-type specific and it allows different types of actions to perform their own initialization beyond those constructors defined in *Action*, such as computing the resource usage set.
- Abstract method *computeResourceUsages* takes *startState*, *startTime*, and *currTime* as parameters, and computes the resource usage set of an action. It returns a list of *ResourceUsage* objects.
- Abstract method *getCurrState(currTime)* calculates the current system state given the current time. The calculation is based on *startState*, *startTime*, and *currTime*. This method returns a *SystemState* object representing the status of a CPS entity at time *currTime*.
- Abstract method *adjustedResourceUsage(startState,startTime,endTime,otherPars)* returns the resource usage set of an action if it is adjusted through its attributes. However, it makes no change to the action. This method will be used by a transaction when it is doing action adjustment in order to get a prediction of resource usages by an action.

Each sub-class of *Action* inherits fields and methods from *Action*. However, they may define additional ones. For example, *IncreaseSpeedAction* and *DecreaseSpeedAction* have a

property *targetSpeed*, and *TurnLeftAction* and *TurnRightAction* have a property *turningAngle*. As for methods, all sub-classes must implement action-type-specific abstract methods, i.e., *computeResourceUsages*, *updateFields*, *getCurrState*, and *adjustedResourceUsage*.

6.3.3 Transaction

The implementation mechanism for transactions is similar to that for actions. An abstract class *Transaction* is defined, and several sub-classes are defined to represent different types of transactions (Figure 6.5).

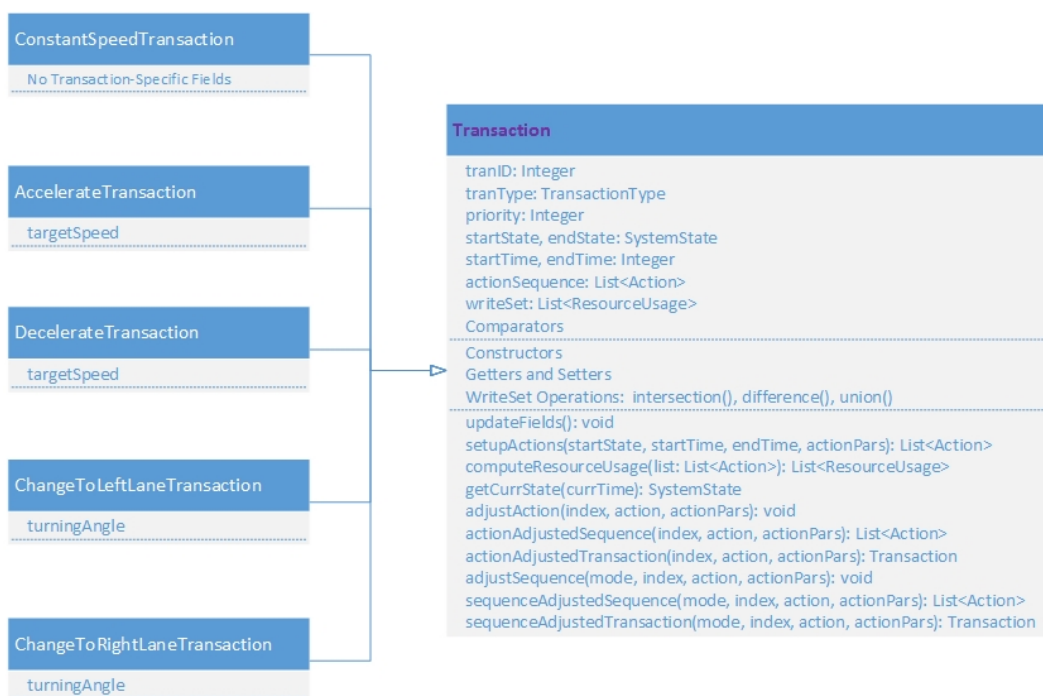


Figure 6.5: Transaction

Similar fields are defined for *Transaction*, such as *tranID*, *tranType*, *priority*, *startState*, and *startTime*. Specially, the same levels of *priority* are defined and it is used in transaction-level two-phase commit processing. Level 4 indicates a normal transaction to be scheduled for execution, level 3 is a conflict-resolving or cooperative transaction, level 2 is an exception handling transaction, and level 1 is an emergency-handling transaction.

Besides, an *actionSequence* field is defined to represent the sequence of actions in a

transaction, and the *writeSet* field indicates the resource usage set of a transaction, which is a union of resource usage sets of its actions.

As for methods, several abstract methods are defined. These methods are transaction-type-specific and thus require each sub-class of *Transaction* to implement them.

- *updateFields()* initializes transaction fields such as *writeSet* and *actionSequence*, which are not initialized by constructors defined in *Transaction*.
- *setupActions(startState, startTime, endTime, actionPars)* initializes actions in *actionSequence*.
- *computeResourceUsage(list:List>Action_i)* computes *writeSet* given a list of actions.
- *getCurrState(currTime)* calculates the current system state given the current time. The calculation is actually carried out in an action whose execution time period *currTime* is within.
- *adjustAction(index, action, actionPars)* performs action adjustment to a transaction (Section 5.3.5). The action at *index* in the *actionSequence* is replaced with the given *action* initialized with *actionPars*. This method makes actual changes to the underlying transaction.
- *actionAdjustedSequence(index, action, actionPars)* performs action adjustment to a transaction and returns the action sequence after adjustment. The difference from *adjustAction* is that this method doesn't change the underlying transaction, but returns a new sequence of actions.
- *actionAdjustedTransaction(index, action, actionPars)* performs action adjustment to a transaction and returns a new transaction. This method also doesn't change the underlying transaction.
- *adjustSequence(mode, index, action, actionPars)* performs sequence adjustment to a transaction (Section 5.3.5). There are three different modes. If mode is -1, it prepends

the given *action* to the action sequence; if mode is 1, *action* is appended to the action sequence; if mode is 0, *action* is inserted at *index* in the action sequence. *actionPars* are parameters used to initialize *action*. This method makes actual changes to the underlying transaction.

- *sequenceAdjustedSequence(mode, index, action, actionPars)* performs sequence adjustment to a transaction and return the action sequence after adjustment. This method doesn't change the underlying transaction, but returns a new sequence of actions.
- *sequenceAdjustedTransaction(mode, index, action, actionPars)* performs sequence adjustment to a transaction and returns a new transaction. This method doesn't change the underlying transaction.

6.4 Cyber-Physical System

Our implementation of a Cyber-Physical System focuses on the transaction processing features. We define four main components: Transaction Generator, Transaction Scheduler, Transaction Executor, and Resource Manager (Figure 6.6).

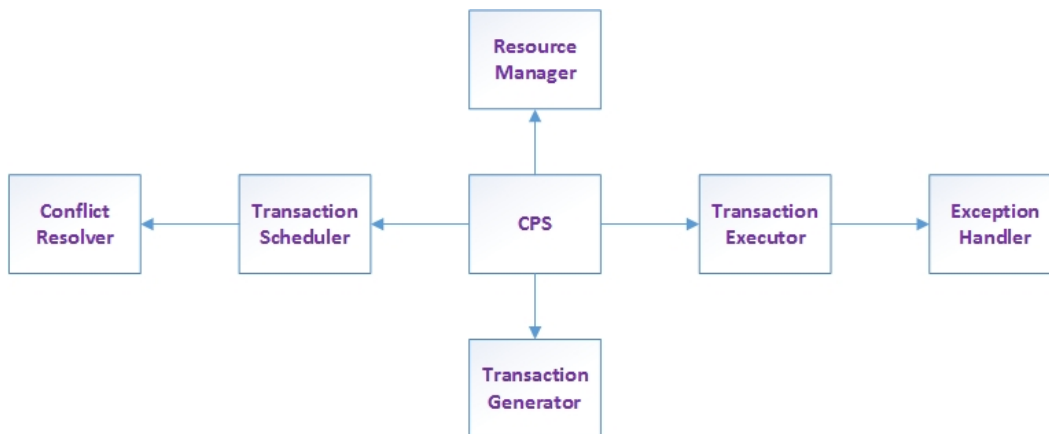


Figure 6.6: Architecture of Cyber-Physical System

The class *CyberPhysicalSystem* represents a Cyber-Physical System or a CPS entity,

and it organizes different components into an independent system that operates on its own. The design of *CyberPhysicalSystem* is highly modularized and allows replacing a component with another as long as they provide the same interface. The main fields defined in *CyberPhysicalSystem* include:

- *cpsID* and *comID* are the CPS and component identification of a CPS entity. *cpsID* identifies a CPS entity in a CPS Network, and *comID* identifies the CPS entity as a parent component compared with sub-components such as Transaction Scheduler and Executor.
- *startState* and *endState* are the initial and goal system states of a CPS entity respectively.
- *startTime* and *endTime* are the start and end execution times of a CPS entity respectively.
- *scheduleQueue*, *executeQueue*, *abortQueue*, *finishQueue* and *failQueue* are queues of transactions used for different purposes.
- *tranGenerator* is the Transaction Generator component that generates new transactions.
- *tranScheduler* is the Transaction Scheduler component that schedules transactions in the pre-commit phase.
- *tranExecutor* is the Transaction Executor component that executes transactions in the commit phase.
- *resrManager* is the Resource Manager component that manages resource reservations of the host CPS entity.
- *inQueue* is a message queue used by the Communication Network to deliver messages targeting this CPS entity.

- *distMap* is a map from component identities (*comID*) to incoming message queues of components. This mapping is used by *InternalMessageDistributor* to deliver messages to different components.
- *distributor* is an *InternalMessageDistributor* that takes care of internal message distribution.
- *Network* is the Communication Network.
- *PPStrategy* indicates the strategy used to resolve PP Conflict (will be covered when discussing Transaction Scheduler).
- *PCStrategy* indicates the strategy used to resolve PC Conflict (will be covered when discussing Resource Server).
- *simulationMode* is the simulation mode applied by *CPSNetwork*. Two simulation modes are supported corresponding to how environmental resources are managed: *centralized* or *distributed*.
- *running* is the status flag indicating whether a CPS entity is running.
- *currentSystemState* is the current status of a CPS entity.
- *currentTransaction* is the current transaction that is being executed.

Besides constructors, getter and setter methods, *CyberPhysicalSystem* defines methods to control the running of a CPS entity and to collect statistical information related to transaction processing algorithms.

The control methods include *start*, *update*, and *stop*.

- In the *start* method, components of a CPS entity are started if they run as separate threads. The initialization of components takes place in the constructor.

- The *update* method takes the current time as the parameter and updates the status of a CPS entity. This method triggers operations of Transaction Generator, Scheduler and Executor components, which cooperate to process transactions.
- The *stop* method terminates the operations of components. This method is called when the simulation is completed.

Statistics methods collect information such as how many transactions are generated, scheduled, and executed, how many transactions are adjusted, and how many successful adjustments happen. We will discuss this statistical information in the next chapter when we show and analyze simulation results.

6.4.1 Transaction Processing Flow

The transaction processing flow of a CPS entity is shown in Figure 6.7. The figure shows how different components interact with each other to process transactions. Since we have already discussed transaction processing in Chapter 5, we will not go over all those details here.

6.4.2 Transaction Generator

The function of the Transaction Generator is to create new transactions that will be scheduled and executed by Transaction Scheduler and Executor respectively. The implementation of class *TransactionGenerator* is shown in Figure 6.8.

There are three modes the generator follows to create a new transaction.

The first is **automatic** mode, where the generator automatically picks the most suitable transaction to achieve the goal state (i.e., *goalState* in Figure 6.8). The selection of a transaction is based on a score that is calculated based on the similarity between the end system state of a transaction and the *goalState*. Higher the score, better is the chance a transaction has to reach the goal. Hence, the transaction with the highest score is selected.

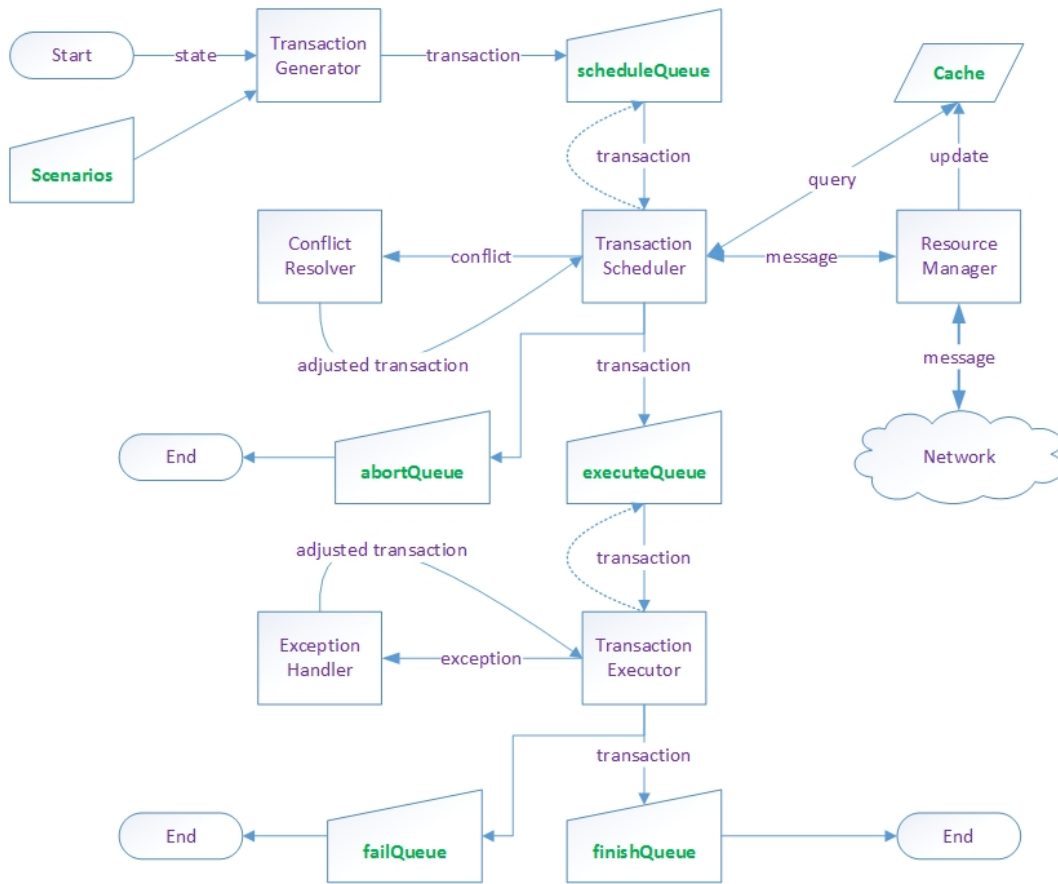


Figure 6.7: Transaction Processing Flow



Figure 6.8: Transaction Generator

The corresponding method for this mode is: *generateTransactionDefault(startState,goalState,startTime,endTime):void*.

The second is **scenario** mode, where a sequence of transactions is specified in a scenario file for the generator to create. Each scenario in the file specifies the type, the execution time duration, and action parameters of a transaction. For example, “AccelerateTransaction, 5000, 5.0” indicates creating an *AccelerateTransaction* with a target speed 5.0 meters per second, and the execution time lasts for 5000 milliseconds. The method for this mode is *generateTransactionScenario(startState,startTime):void*, and the scenario file is read when the Transaction Generator is initialized.

The third mode is **passive** mode. In the passive mode, a new transaction is generated upon requests. For example, Transaction Scheduler may need to create new transactions to resolve transaction conflicts, and Transaction Executor may create new transactions to handle exceptions. The request is made through method *generateTransaction(type,startState,startTime,endTime,pars):void*. Here, *type* indicates the type of transaction to be generated, and *pars* is an object array that contains parameters to initialize the transaction.

Once a transaction is generated, it is placed into *scheduleQueue* for further processing.

6.4.3 Transaction Scheduler

The Transaction Scheduler schedules transactions in *scheduleQueue* that have been generated by Transaction Generator. Once a transaction is scheduled, it is placed in *executeQueue* waiting for Transaction Executor to execute it. Otherwise, it is put in *abortQueue*, which means the transaction is aborted. The class diagram is shown in Figure 6.9. Since the scheduling process is different for different resource management mechanisms, *TransactionScheduler* is defined as an abstract class, and two sub-classes inherit it: *CentralizedTransactionScheduler* and *DistributedTransactionScheduler*.

Both *CentralizedTransactionScheduler* and *DistributedTransactionScheduler* are implemented as threads, and they keep checking whether new transactions are placed in *scheduleQueue*.

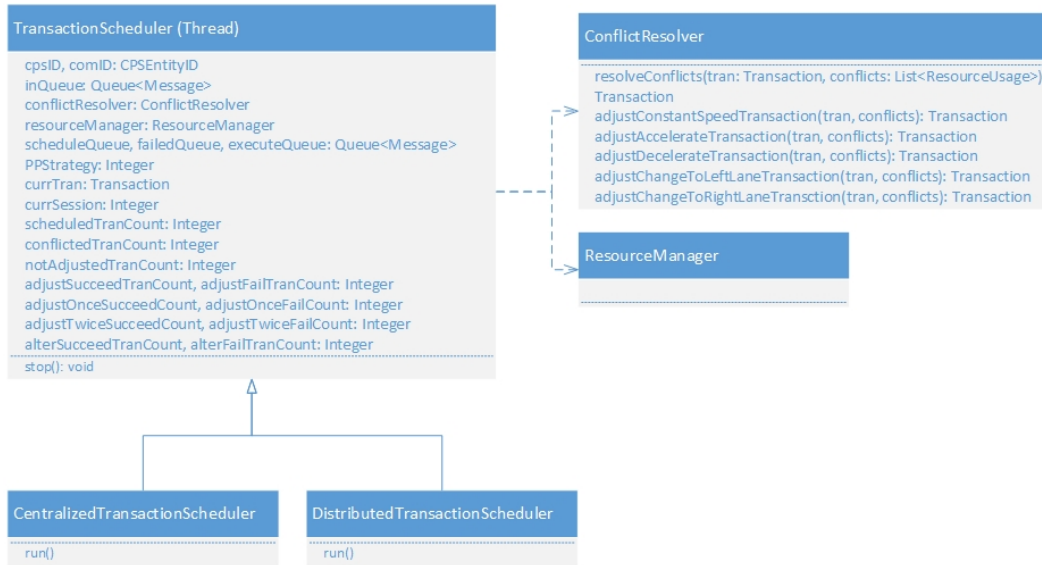


Figure 6.9: Transaction Scheduler

If transactions exist in *scheduleQueue*, a transaction is retrieved and scheduled. Since we have discussed the scheduling process carried out by a scheduler as a client role in both centralized and distributed resource management environments in Section 5.4, we will skip the discussion of the scheduling process here.

In the scheduling process, the scheduler makes resource reservations for transactions through the Resource Manager component (class *ResourceManager*). Resource Manager component works as the portal for resource usage negotiation. We will cover more about Resource Manager later.

When transaction conflicts are detected, they need to be resolved. *PPStrategy* (Figure 6.9) indicates the strategy used by Transaction Scheduler to resolve PP Conflict (Section 5.3.1). However, PC Conflict is detected and resolved by the Resource Server because only the Resource Server can decide whether to allow transaction preemption and when to revoke existing resource reservations. There are three levels of strategies defined for *PPStrategy*: 0, 1, and 2. Level 0 means the strategy is not set and a conflicting transaction is aborted in default, level 1 indicates Win-Lose strategy, and level 2 indicates Win-Win strategy.

When potential transaction conflicts are found, if Win-Win strategy is specified for resolving PP Conflict, Transaction Scheduler refers to **Conflict Resolver** component (class *ConflictResolver*) to figure out how to adjust the transaction in order to avoid potential conflicts. The Conflict Resolver analyzes existing transaction conflicts, evaluates characteristics of different transactions, and comes up with adjustment solutions to resolve conflicts for each type of transactions. In the current implementation, a transaction can be adjusted at most twice. After that, an alternative transaction is created to replace the current one since the potential conflicts can't be resolved by previous two adjustments. The Conflict Resolver component also takes care of creating alternative transactions.

TransactionScheduler keeps a set of counters that measure the performance of the scheduling algorithm regarding its capability to resolve potential conflicts.

- *scheduledTranCount* counts how many transactions have been successfully scheduled.
- *conflictedOriginalTranCount* counts how many original transactions have had potential conflicts with others. An original transaction is a transaction that has not been adjusted.
- *notAdjustedTranCount* counts how many conflicting transactions were not adjusted (when using Win-Lose strategy).
- *adjustSucceedTranCount* and *adjustFailTranCount* count how many conflicting transactions that were adjusted succeeded in avoiding potential conflicts, and how many failed.
- *adjustOnceSucceedCount* and *adjustOnceFailCount* count how many conflicting transactions that were adjusted only once succeeded in avoiding potential conflicts, and how many failed.
- *adjustTwiceSucceedCount* and *adjustTwiceFailCount* count how many conflicting transactions that were adjusted twice succeeded in avoiding potential conflicts, and

how many failed.

- *alterSucceedTranCount* and *alterFailTranCount* count how many alternative transactions that were created to replace conflicting transactions succeeded in resolving conflicts, and how many failed.

6.4.4 Transaction Executor

The Transaction Executor (Class *TransactionExecutor* in Figure 6.10) executes transactions in *executeQueue* that are scheduled by Transaction Scheduler. The executor simulates operations of the physical subsystem by updating the status of a CPS entity given the transaction currently being executed and the current system time. When exceptions occur, Exception Handler (Class *ExceptionHandler*) handles them in order to maintain a consistent CPS Network, such as aborting or adjusting the current transaction. When a transaction is successfully executed, it is placed in *finishQueue*; otherwise it is placed in *failQueue*.

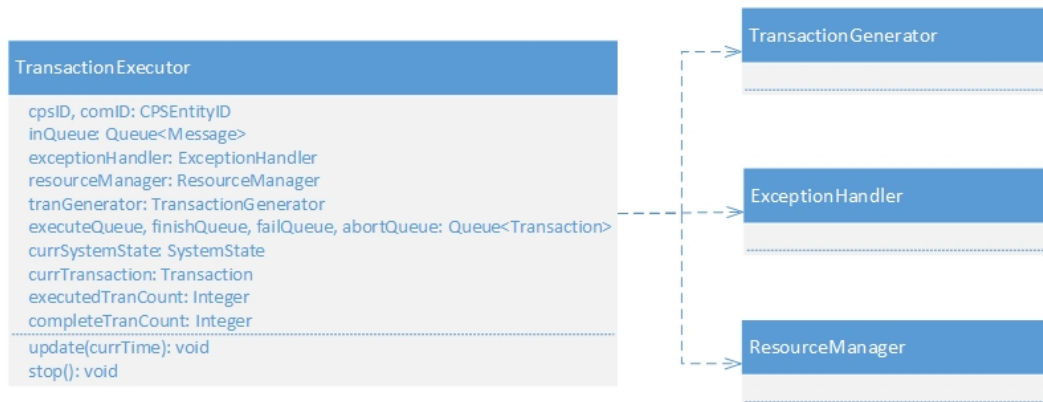


Figure 6.10: Transaction Executor

However, our simulation concentrates on resolving conflicts in the pre-commit phase, and we assume that the execution of a scheduled transaction always succeeds without exceptions. Thus, the integration of Transaction Executor and Exception Handler are neglected in the current simulations, and what Transaction Executor does is to update the system state of a CPS entity at fixed time interval (method *update(currTime):void*).

6.4.5 Resource Manager

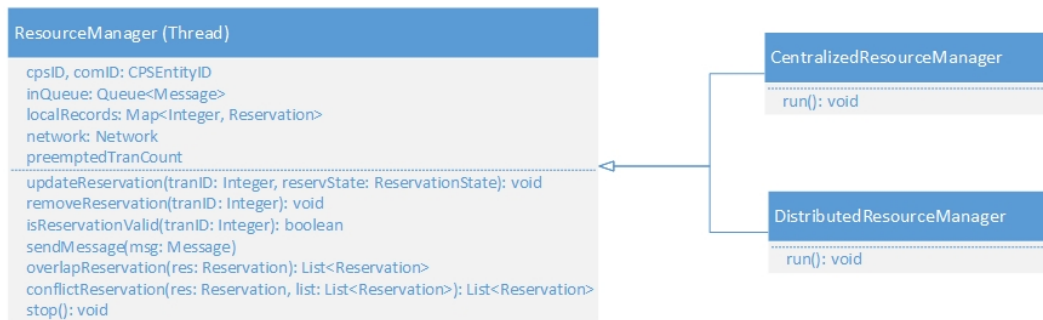


Figure 6.11: Resource Manager

Resource Manager (Class *ResourceManager* in Figure 6.11) acts as an agent for a CPS entity to coordinate resource usages with other CPS entities. It tracks resource reservation states of transactions and keeps a local record of existing reservations (*localRecord*).

If there is a central Resource Server (centralized environment), Resource Manager works as a resource client and speaks with the server directly. If there is no central server, then Resource Manager plays both resource client and server roles.

- When reserving resources, it works as a client and communicates with Resource Managers of other CPS entities.
- When handling resource reservation requests from other CPS entities, it works as a server and checks potential transaction conflicts for incoming reservation requests.

Class *ResourceManager* is defined as an abstract class. Two sub-classes, *CentralizedResourceManager* and *DistributedResourceManager*, inherit *ResourceManager* and work in centralized and distributed resource management environment respectively. The implementation for *CentralizedResourceManager* is straightforward and it includes the following basic functions:

- Forward resource-related messages from other components to the CPS Network. Some pre-processing jobs are performed before sending those messages out, such as adding, updating, and removing local records.

- Keep track of reservation states of the host's transactions. Possible reservation state includes *Pending*, *PendingACK*, *Succeed*, *Failed*, *Revoked*, and *Expired* (will be covered later when discussing Resource Server).
- Support internal and external checking in the commit phase to detect potential exceptions.

The implementation for *DistributedResourceManager* is more complex since it is a combination of resource client and server roles. From functionality viewpoint, it is the same as the union of *CentralizedResourceManager* and *ResourceServer*, and we will discuss this along with *ResourceServer* later.

6.5 Resource Server

The implementation of the Resource Server (Class *ResourceServer*) is shown in Figure 6.12. It is used only in the centralized resource management environment. The Resource Server for the distributed environment is implemented by *DistributedResourceManager*. Here, we go through only the implementation of *ResourceServer*, and the same implementation strategy is applicable to *DistributedResourceManager*.

6.5.1 Tasks of a Resource Server

In the centralized environment, all CPS entities negotiate with Resource Server to reserve resources for their transactions. Resource Server has three main tasks.

The first task is to maintain a record of all resource reservations. Each reservation is represented by a *Reservation* object (Figure 6.13) and represents a resource reservation of a transaction. *ResourceServer* uses three data structures to keep reservations in different states: *pendingWithoutACK*, *pendingWithACK*, and *reservations*. Possible reservation states include:

- *Pending*: a new reservation that hasn't been processed yet. This type of reservations are stored in *pendingWithoutACK*.

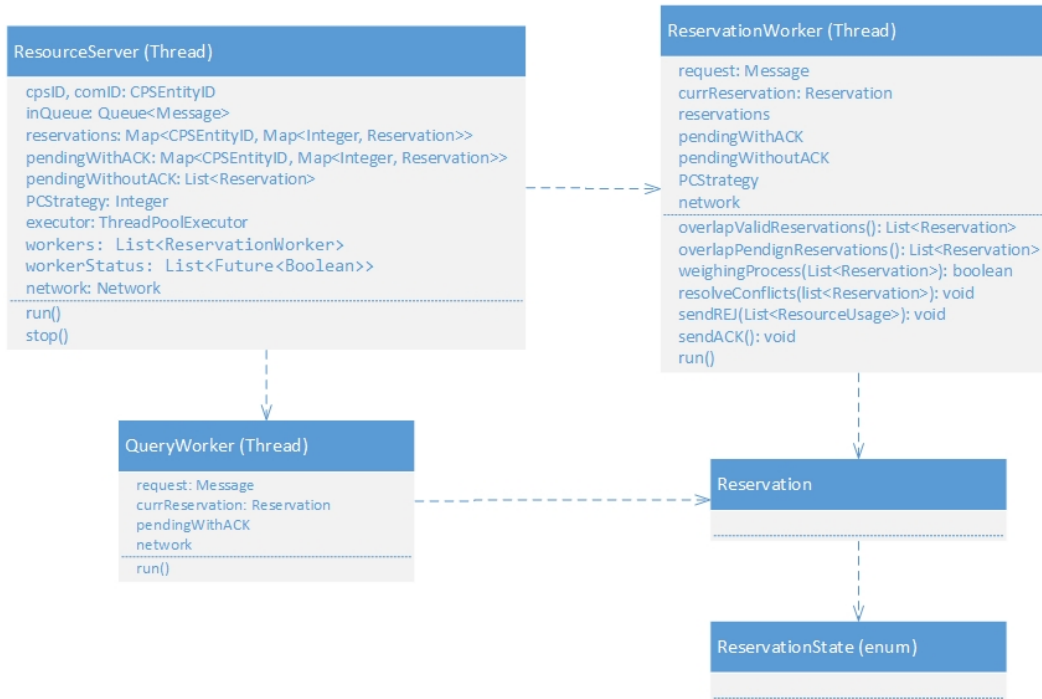


Figure 6.12: Resource Server

- *PendingACK*: a reservation that has been processed, and an acknowledgement has been sent by the server, but a final confirmation from the client is not received yet. This type of reservations are stored in *pendingWithACK*.
- *Succeed*: when a *PendingACK* reservation is confirmed by the client, the reservation becomes *Succeed* and is stored in *reservations*.
- *Failed*: when a *Pending* reservation can't be made by the server due to transaction conflicts, it becomes *Failed*.
- *Revoked*: a *Succeed* reservation that has been revoked by the server (due to transaction preemption).
- *Expired*: a *Succeed* reservation that has expired.
- *Query*: a query reservation, which actually doesn't reserve resources, but query resource usage status.

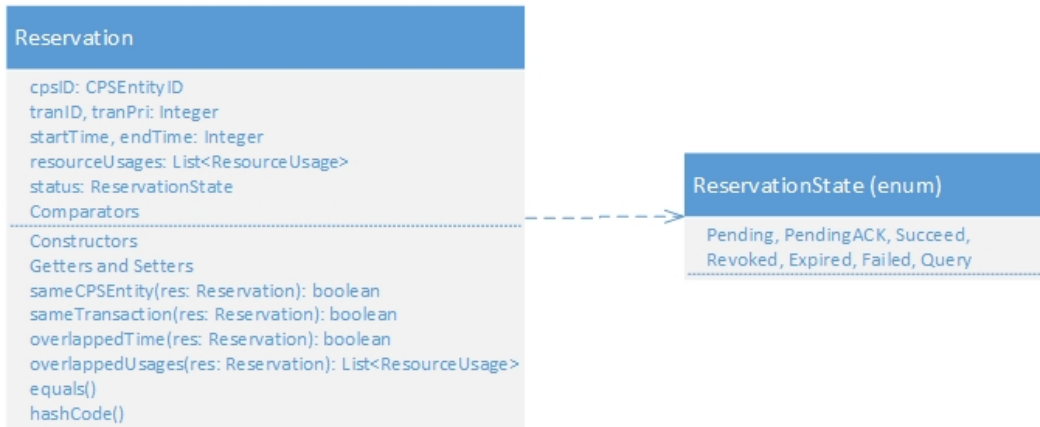


Figure 6.13: Reservation and Reservation State

The second task is to process resource reservation requests. We have discussed how a Resource Server processes an incoming reservation request in Section 5.4, so we won't go through each step here. In the implementation, reservation requests are processed by *ReservationWorker*, and each reservation request is handled by a separate thread. When a new reservation request comes in, a *Reservation* object is created and saved in *pendingWithoutACK*. Then a new *ReservationWorker* thread is initialized to handle the request, and operations are invoked to check potential resource conflicts against reservations stored in *pendingWithACK* and *reservations*.

- If no conflicts are found, then the reservation can be made and an acknowledgement is sent to the client and the reservation is moved to *pendingWithACK*. When a confirmation is received from the client, the reservation becomes *Succeed* and is moved to *reservations*.
- If conflicts are found or if no confirmation is received after the acknowledgement, then a rejection message is sent and the reservation becomes *Failed* and is removed from *pendingWithoutACK*.

When PC Conflict is found, *PCStrategy* is applied to resolve the conflict. There are three levels defined. Level 0 means an unset strategy and no transaction preemption will

be allowed, level 1 means transaction preemption is allowed on reservations with *Pending* state, and level 2 means transaction preemption is allowed on reservations that are in one of *Pending*, *Succeed*, and *PendingACK* states.

The third task is to process resource usage queries. Query requests check potential transaction conflicts by providing the pre-Write set of a transaction. Each time a query request is received, a *QueryWorker* thread is created to process the request. *QueryWorker* checks potential transaction conflicts between the given transaction and those recorded in *pendingWithACK* and *reservations*. Checking results are sent back to the client.

6.5.2 Message

The communication between the Resource Server and clients is carried out through messages. Types of messages exchanged between the Resource Server and clients include:

- *REQ*: “Request of New Reservation” sent from client to server.
- *ACK*: “Acknowledge Pending Reservation” sent from server to client.
- *REJ*: “Reject Pending Reservation” sent from server to client.
- *CAN*: “Cancel Pending Reservation” sent from client to server.
- *CON*: “Confirm Pending Reservation” sent from client to server.
- *REV*: “Revoke Valid Reservation” sent from server to client.
- *CHK*: “Check for Resource Usage Conflicts” sent from client to server.
- *RES*: “Response to CHK Message” sent from server to client.

The implementation of message classes is shown in Figure 6.14, which includes four classes: *Message*, *MessageType*, *MessageHeader*, and *MessageBody*.

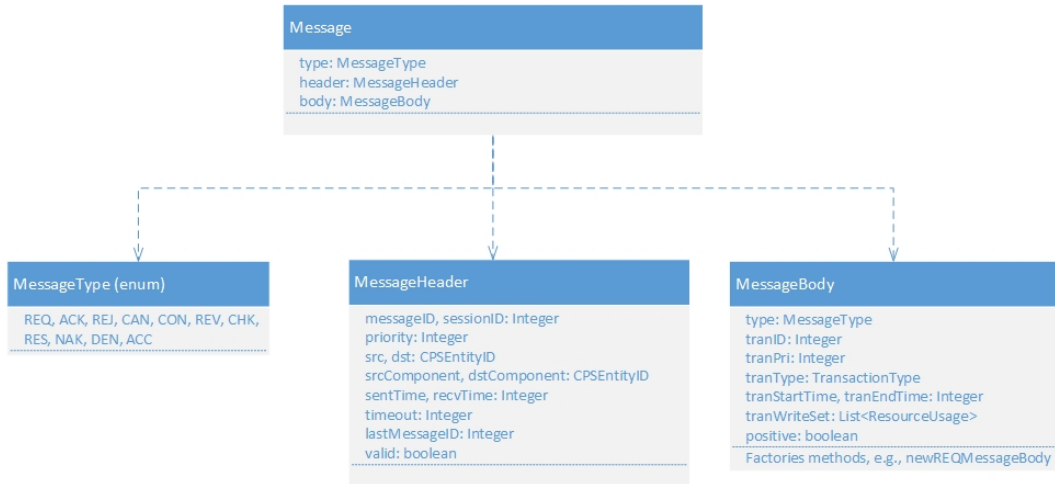


Figure 6.14: Message

6.5.3 Reservation State Machine

The relationship between messages and reservation states is shown by the Reservation State Machine in Figure 6.15

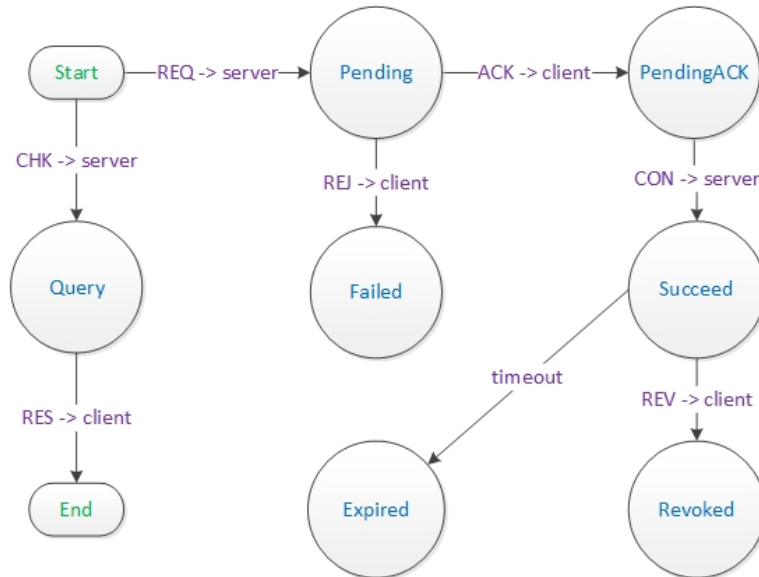


Figure 6.15: Reservation State Machine

6.6 Communication Network

The Communication Network delivers a message from its sender to its recipient. However, the delivery process is not our concern in the simulation, and we don't need a full-scaled network with different protocols and various features. Thus, we use a straightforward approach to simulate the Communication Network, as shown in Figure 6.16, and the distribution flow of a message is illustrated in Figure 6.17.

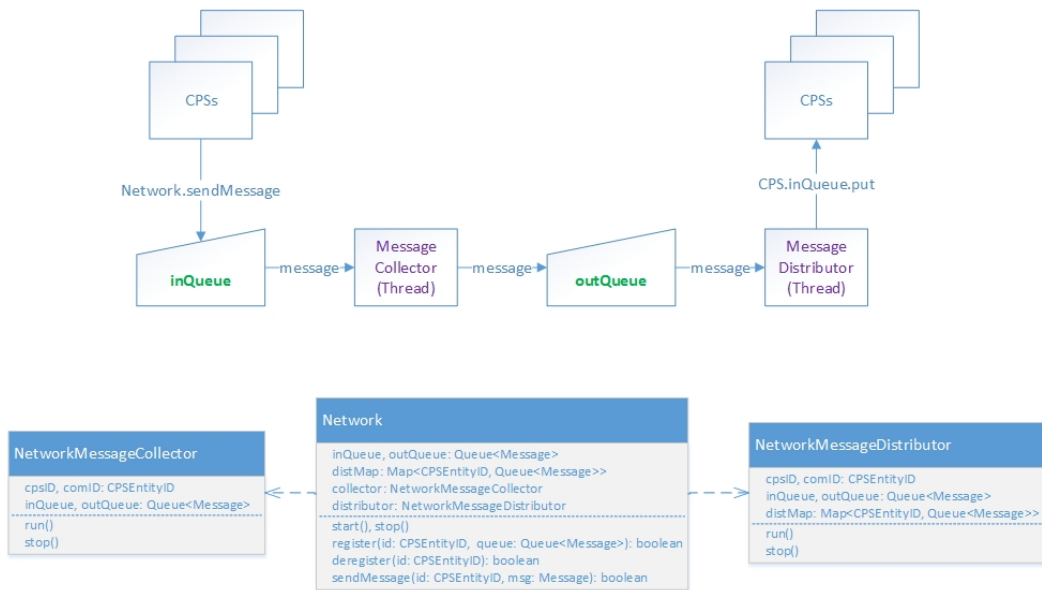


Figure 6.16: Communication Network

There are two levels of message transmission implemented: external and internal.

In **external** message transmission, each CPS entity (including Resource Server, if any) sends a message by calling the `sendMessage` method of `Network`, and then the message is placed in the incoming message queue `inQueue` of `Network`. Thread `NetworkMessageCollector` processes incoming messages in `inQueue` by simulating delays, errors, etc., and then moves them to `outQueue`. Thread `NetworkMessageDistributor` takes message from `outQueue` and puts them in `inQueues` of their target CPS entities.

In **internal** message transmission, when a message arrives at the `inQueue` of a CPS entity, it is handled by `InternalMessageDistributor`, who delivers the message to its target

component.

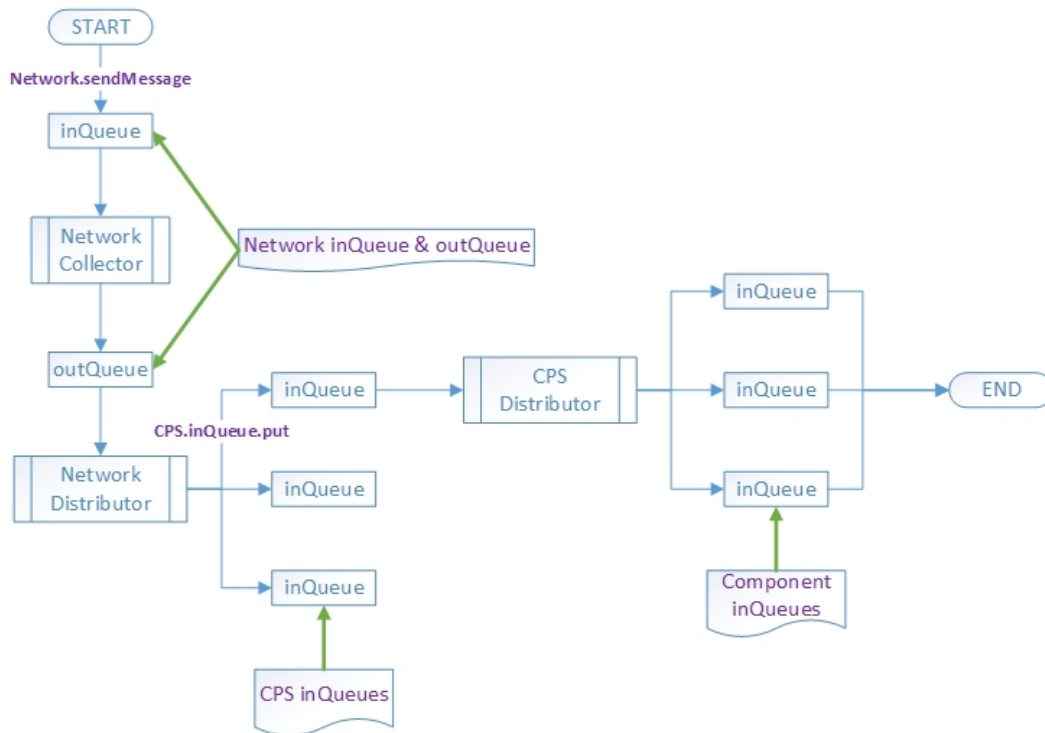


Figure 6.17: External and Internal Message Transmission

To make both external and internal message transmission work, each CPS entity must register their *cpsIDs* and *inQueues* in *Network*, and each component must register their *comIDs* and *inQueues* in their parent *CyberPhysicalSystem*. The mapping from *cpsID* or *comID* to *inQueue* is recorded in a map (*distMap* in both *Network* and *CyberPhysicalSystem*). Besides, each message should specify its source and destination CPS entity and component (*src*, *dst*, *srcComponent*, and *dstComponent* in Figure 6.14).

Two types of delivery are supported: uni-cast, and broadcast. Uni-cast is one-to-one message delivery, and broadcast is one-to-all. For broadcast, the *dst* and *dstComponent* will be *BROADCAST* defined in class *CPSNETConfiguration*.

6.7 CPS Network

Having discussed components in the lower three levels of Figure 6.1, we now present the top level: *CPSNetwork*. Class *CPSNetwork* organizes entities in the 2nd level together into a simulation platform and serves as the simulation entry point. It takes in different configuration files, initializes environment variables, creates different entities, and runs a time-based simulation to update status of all CPS entities in the network. The architecture and implementation of *CPSNetwork* are shown in Figure 6.18, and the simulation flow followed by *CPSNetwork* is illustrated by Figure 6.2.

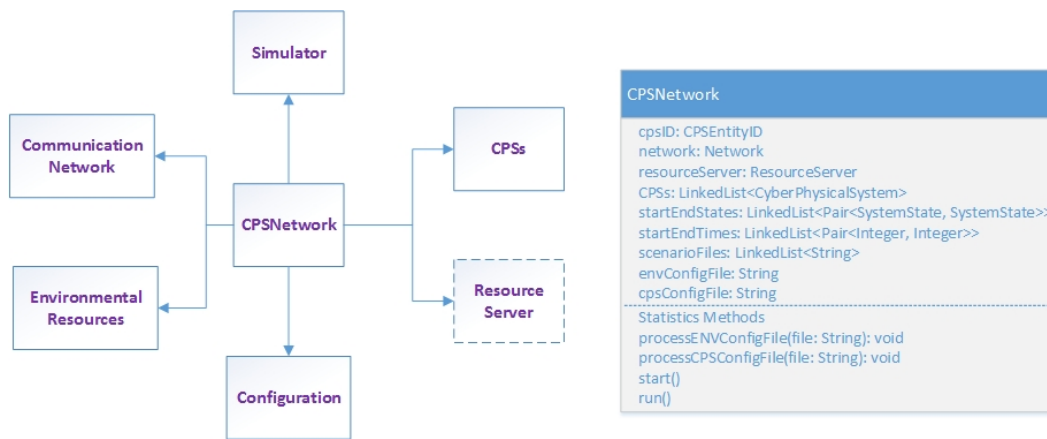


Figure 6.18: CPSNetwork

6.7.1 Configuration

Two levels of configuration are provided by *CPSNetwork*.

The first level is configuration of the CPS Network. In this level, simulation-related properties are initialized, such as number of CPS entities, time and length unit, global clock, simulation duration, simulation mode (centralized or distributed), and logging and result directories. The configuration file for this level is specified by *envConfigFile*. The file is processed by method *processENVConfigFile*, and related properties are kept in class *CPSNETConfiguration*.

The second level is configuration of entities in the network. These entities include environmental resource, Resource Server, and CPS entities. In this level, entity-related properties are initialized.

- For environmental resource, properties such as number of lanes in the simulated road and the length and width of a road segment are specified in *CPSNETConfiguration*.
- For Resource Server, server identity, broadcast address, and conflict resolving strategy for PC Conflict are specified in *CPSNETConfiguration*.
- For CPS entity, start and end system states, and scenario files are specified in file *cpsConfigFile*, which is processed by method *processCPSConfigFile* of *CPSNetwork*. Other properties such as PP Conflict resolving strategy, the maximum speed and acceleration of a car, and the waiting time for message response are defined in *CPSConfiguration*.

We will show how the configuration is performed and the contents of the configuration files in the next chapter when we go through the simulation experiments.

6.7.2 Time-Based Simulation

As the simulation entry point (Figure 6.2), when started, *CPSNetwork* first reads and processes the configuration files, then configures the simulation environment. After that, the Communication Network (Class *Network*), the Resource Server (Class *ResourceServer*, if centralized simulation mode is specified), and all CPS entities (Class *CyberPhysicalSystems*) are initialized. After initialization is done, all entities are started by calling their *start* methods. Since they are threads, they run independently, and the interactions among them are through sending and receiving messages.

The simulation of CPS entities is time-based. To fulfill this, a global clock is simulated to provide time service. The time interval for updating statuses of CPS entities and the simulation time duration guide the simulation process, and both are specified in *CPSNETCo*

nfiguration. In every time interval, the *update* method of each CPS entity is called to update its status. This method would trigger operations of Transaction Generator, Scheduler, and Executor, which further trigger operations of other components of a CPS entity, e.g., Conflict Resolver. If transaction generation follows the *scenario* mode, when the last transaction is executed by a CPS entity, its operation is terminated. Otherwise, in the *automatic* mode, transactions are generated automatically, and operations of each CPS entity go on as long as the simulation duration is not ended. In either case, the simulation duration should be given a value that is larger than the end execution time of the last transaction among all CPS entities.

When simulation time duration is reached, the simulation is terminated by terminating the Resource Server, CPS entities, and Communication Network entities in order by calling their *stop* methods. Then statistical information are collected from all these entities through their statistical methods.

6.7.3 Statistical Information

CPSNetwork collects statistical information from all entities in the 2nd level, which in turn get the information from components in the 3rd level, as illustrated in Figure 6.19.

Besides these statistical information, operation logs are kept for each thread component and entity. Through these logs, the behavior and performance of each component or entity can be observed.

In next chapter, more details will be covered about statistical information and operation logs. We will discuss how to use them to evaluate the simulation platform and the transaction model.

6.8 Summary

In this chapter, we discuss how the simulation platform **CPSNET** is designed and implemented in four levels (Figure 6.1). The *CPSNetwork* class works as the simulation entry

	A	B	C
	Counter	Source Entity or Component	Description
1	GeneratedTranCount	TransactionGenerator	Number of transactions that are generated.
2	ConflictedOriginalTranCount	TransactionScheduler	Number of original transactions in all CPS entities that have potential conflicts with others.
3	NotAdjustedTranCount	TransactionScheduler	Number of conflicting transactions in all CPS entities that are not adjusted.
4	AdjustOnceTranCount	TransactionScheduler, ConflictResolver	Number of conflicting transactions that have been adjusted at least once.
5	AdjustTwiceTranCount	TransactionScheduler, ConflictResolver	Number of conflicting transactions that have been adjusted at least twice.
6	AlternativeTranCount	TransactionScheduler, ConflictResolver	Number of alternative transactions that have been created to replace conflicting transactions.
7	AdjustSucceedTranCount	TransactionScheduler	Number of adjusted transactions that succeed to resolve conflicts..
8	AdjustFailTranCount	TransactionScheduler	Number of adjusted transactions that fail to resolve conflicts.
9	AdjustOnceSucceedCount	TransactionScheduler	Number of once-adjusted transactions that succeed to resolve conflicts.
10	AdjustOnceFailCount	TransactionScheduler	Number of once-adjusted transactions that fail to resolve conflicts.
11	AdjustTwiceSucceedCount	TransactionScheduler	Number of twice-adjusted transactions that succeed to resolve conflicts.
12	AdjustTwiceFailCount	TransactionScheduler	Number of twice-adjusted transactions that fail to resolve conflicts.
13	AlterSucceedTranCount	TransactionScheduler	Number of alternative transactions that succeed to resolve conflicts.
14	AlterFailTranCount	TransactionScheduler	Number of alternative transactions that fail to resolve conflicts.
15	ScheduledTranCount	TransactionScheduler	Number of transactions that are successfully scheduled.
16	ExecutedTranCount	TransactionExecutor	Number of transactions that are executed.
17	CompleteTranCount	TransactionExecutor	Number of transactions that are executed completely.
18	IncompleteTranCount	TransactionExecutor	Number of transactions that are executed partially.
19	PreemptedTranCount	ResourceManager	Number of transactions that have are preempted.
20	ConflictReservationCount	ResourceServer, ReservationWorker	Number of reservations that have conflicts with others.
21	NumberOfMessages	Network	Number of messages that have been sent through the network.
22			

Figure 6.19: Statistical Information of CPSNetwork

point and it organizes different entities in the second level together. The second level is composed of entities with different functions, such as the Resource Server (Class *ResourceServer*), the Communication Network (Class *Network*), and CPS entities (Class *CyberPhysicalSystem*). These entities are composed of components in the third level, and the parent entity’s functionality is fulfilled by these components. In the bottom level, basic classes are defined to provide different abstractions for resources, messages, actions, and transactions.

We show the design and implementation of different classes in different levels of the platform, and discuss their functions and operation flows. Especially, we show how a transaction is processed in each CPS entity and the functions of different components with respect to transaction processing (Figure 6.7), how internal and external message distribution is performed (Figure 6.17), and how a simulation is carried out by the *CPSNetwork* class (Figure 6.2). In the next chapter, we will present simulation experiments and results using the simulation platform to evaluate our transaction model. .

Chapter 7

Simulation and Results

7.1 Introduction

In the previous chapter, we presented the CPSNET simulation platform. Specifically, we show the design and implementation of different types of entities (*CyberPhysicalSystem*, *Network*, and *ResourceServer*) in a CPS Network (*CPSNetwork*), and their functions and roles in the processing of transactions. We also describe how a simulation is carried out in a *CPSNetwork* (Figure 6.2). In this chapter, we show how to use the simulation platform to simulate and verify the transaction model. The verification includes two aspects: the simulation platform itself and the transaction model.

To verify the simulation platform, we show that every entity in a CPS Network and every component in a CPS entity work as expected, and the transaction processing flow in Figure 6.7 is followed by every corresponding entity and component. We prepare one group of test for this verification (Group 0). It consists of only one CPS entity which executes 26 transactions sequentially. The set of transactions that the CPS entity has to execute are specified in a scenario file. By examining the statistical information obtained through *CPSNetwork* and *CyberPhysicalSystem*, and the operation logs of different components and entities, our tests for Group 0 shows that:

- Component *TransactionGenerator* generates transactions specified in the scenario file correctly.

- Component *TransactionScheduler* schedules transactions that are generated by *TransactionGenerator* correctly, and the transaction scheduling flow is followed.
- Component *TransactionExecutor* executes transactions that are scheduled by *TransactionScheduler* as expected.
- Component *ResourceManager* handles resource-related messages and maintains resource reservations as expected.
- *ResourceServer* processes resource reservation requests as expected.
- *Network* delivers messages correctly, i.e., all messages are successfully sent and received.
- *CPSNetwork* carries out a simulation as expected, and the simulation always finishes without errors.

To verify the transaction model, we prepare five groups of tests to show that our two-phase commit transaction processing algorithm helps reduce resource usage conflicts between transactions. In each group, two CPS entities with one or two transactions are simulated to show how a certain type of transaction is adjusted when it is in conflict with some other transaction. To create the desired transaction conflicts, we pre-define the start system state and the start execution time of each CPS entity. We evaluate different conflict resolving strategies and compare their results to cases where no conflict resolving is applied or a direct abortion is used for a conflicting transaction. These strategies include: Win-Lose, Win-Win, and Transaction Preemption strategies. The conflicts we focus are *PP* and *PC* Conflict (Section 5.3.1). Simulation results show that our conflict resolving strategies are able to resolve conflicts for different types of transactions.

Finally, we perform a comprehensive simulation study (Group 6) with different numbers of CPS entities and transactions. In this group, we first simulate 10 CPS entities concurrently with a total of 188 transactions (each CPS entity has 12 to 26 transactions). Then

we analyze the simulation result, and evaluate the performance using statistical information collected from the simulation. At last, we test the scalability of the simulation platform by increasing the number of CPS entities and transactions separately and check the performance of the simulation platform. This group not only proves our conclusions in previous six groups, but also shows that the simulation platform and the two-phase commit algorithm can support a large load of CPS entities and transactions. The simulation result shows that the application of the two-phase commit algorithm and the conflict resolving strategies improves the productivity and throughput of a schedule (defined later) compared with cases when no conflict resolving is used or direct abortion of a conflicting transaction is applied. It also shows that the Win-Win strategy has a better performance than the Win-Lose strategy in resolving transaction conflicts and maintaining transaction concurrency.

In the following sections, we first describe assumptions we make when simulating the transaction model, and configuration files used in the simulation. Then we define several criteria that evaluate performance of the two-phase commit algorithm and the test cases to be used in each simulation. After that, we go through each simulation group and analyze the simulation result.

7.2 Assumptions and Configurations

In all of the simulation groups, we assume that a centralized *ResourceServer* exists, i.e., centralized resource management environment is applied. While the distributed environment is also supported, the centralized environment is enough for us to test and verify our transaction model. Thus the distributed environment is ignored in the current simulation. Besides, we continue using the autonomous car example for our simulation.

In order to create different situations for transaction processing, we use the *scenario* mode setting in *TransactionGenerator* to generate a list of transactions for each CPS entity according to the scenario file. This setting helps us to monitor the processing of each transaction, and enables us to verify that each component does its job as expected and


```

1# Configuration of Environment of a CPS Network
2# Environment Settings Affect Configurations of CPS Entities, i.e., CPSNET_CPS_CONFIG.con.
3# Format: key=value. key has the same name as the corresponding field in CPSNETConfiguration.
4 LOG_DIR=log
5 RES_DIR=res
6 NUMBER_OF_LANES=5
7
8 ##### GROUP 0 #####
9 # Group 0, Case 1
10 #SIMULATION_GROUP_NAME=GROUP_0
11 #SIMULATION_CASE_NAME=CASE_1
12 #CONFLICT_RESOLVING=1
13 #PP_CONFLICT_STRATEGY=2
14 #PC_CONFLICT_STRATEGY=2
15 #SIMULATION_TIME_DURATION=100000

```

(a) CPSNET_ENV_CONFIG.config

```

31 # Group 6
32 #CPS_ENTITY_1;2.0,0.0,ROAD_1,1,0,null;0;100000;CPS_ENTITY_1_SCENARIO.scenario
33 #CPS_ENTITY_2;2.0,0.0,ROAD_1,5,0,null;0;100000;CPS_ENTITY_2_SCENARIO.scenario
34 #CPS_ENTITY_3;3.0,0.0,ROAD_1,2,0,null;0;100000;CPS_ENTITY_3_SCENARIO.scenario
35 #CPS_ENTITY_4;3.0,0.0,ROAD_1,4,0,null;0;100000;CPS_ENTITY_4_SCENARIO.scenario
36 #CPS_ENTITY_5;3.0,0.0,ROAD_1,3,0,null;0;100000;CPS_ENTITY_5_SCENARIO.scenario
37 #CPS_ENTITY_6;2.0,0.0,ROAD_1,1,5,null;0;100000;CPS_ENTITY_6_SCENARIO.scenario
38 #CPS_ENTITY_7;4.0,0.0,ROAD_1,2,8,null;0;100000;CPS_ENTITY_7_SCENARIO.scenario
39 #CPS_ENTITY_8;2.0,0.0,ROAD_1,3,5,null;0;100000;CPS_ENTITY_8_SCENARIO.scenario
40 #CPS_ENTITY_9;4.0,0.0,ROAD_1,4,8,null;0;100000;CPS_ENTITY_9_SCENARIO.scenario
41 #CPS_ENTITY_10;2.0,0.0,ROAD_1,5,5,null;0;100000;CPS_ENTITY_10_SCENARIO.scenario

```

(b) CPSNET_CPS_CONFIG.config

Figure 7.1: Examples of Configuration Files

transaction conflicts are detected and resolved correctly.

In the two-phase commit transaction processing algorithm, we focus more on the pre-commit phase, and assume that the commit phase always gets carried out as expected (without exceptions). Thus, in each group of simulation, as long as a transaction is scheduled, it is always executed.

For each simulation, there are three types of configuration files used:

1. Simulation environment configuration file: *CPSNET_ENV_CONFIG.config*. This file specifies the following configurations (Figure 7.1a):

- Directories to save simulation results and operation logs: *LOG_DIR* and *RES_DIR*.
- Number of lanes in the simulated road. We assume that the road has 5 lanes. The width of a lane and the length of a road segment (or grid) are fixed and specified in class *CPSNETConfiguration*.
- Simulation group and case names.

- Conflict resolving flag: *CONFLICT_RESOLVING*. This flag affects the operation of the *ResourceServer*. If it is set to 0, the *ResourceServer* sends an ACK message to any resource reservation request. However, it still performs conflict detection and counts the number of conflicting reservations. If it is set to 1, the *ResourceServer* sends an ACK message only when a reservation can be made. This flag is used to enable counting the number of conflicting reservations.
 - PP and PC conflict resolving strategies: *PP_CONFLICT_STRATEGY* and *PC_CONFLICT_STRATEGY*. Each have three levels: 0-2, as we have described in Section 6.4.3 and Section 6.5.1. *PP_CONFLICT_STRATEGY* determines the strategy applied by the *TransactionScheduler* component to resolve PP Conflict, while *PC_CONFLICT_STRATEGY* decides the strategy applied by the *ResourceServer* to determine how transaction preemption should be carried out to resolve PC Conflict. Both strategies are applicable only when conflict resolving is enabled (*CONFLICT_RESOLVING*).
 - Simulation time: *SIMULATION_TIME_DURATION*. This variable determines the time duration for the simulation. Since the platform has its own global clock to serve the time, this variable doesn't indicate the real-world time length, but number of ticks for the global clock to run.
2. CPS entity configuration file: *CPSNET_CPS_CONFIG.config* (Figure 7.1b). This file specifies CPS entities to be simulated in each group: their identities, start and end system states, operation times, and scenario files used to generate transactions.
 3. Scenarios files. For each CPS entity specified in *CPSNET_CPS_CONFIG.config*, there should be a corresponding scenario file. Each scenario file specifies a list of transactions to be generated by the *TransactionGenerator* component, e.g., Figure 7.3a.

7.3 Performance Evaluation Criteria

To measure the performance of transaction processing and conflict resolving algorithms proposed in Chapter 5, we develop several criteria to evaluate those algorithms. As we know, transactions from CPS entities in a CPS Network form a schedule. These evaluation criteria use the schedule as a basis to quantize the algorithm performance.

Assume that we have a schedule H consisting of transactions from n CPS entities in a CPS Network, and $[t_s, t_e)$ ($t_s < t_e$) is the time period from when the first transaction is triggered to when the execution of the last transaction is finished. During this period, tri_count is the number of transactions triggered, sch_count ($sch_count \leq tri_count$) is the number of transactions scheduled (pre-committed), and exe_count ($exe_count \leq sch_count$) is the number of transactions executed (committed). When scheduling transactions, $conf_count$ is the number of transactions found conflicting with some other transaction. When executing transactions, $excp_count$ is the number of transactions detected to have potential exceptions. Out of exe_count transactions that have been executed, $comp_count$ are completed and others are incomplete.

Schedule Throughput. This category includes three criteria: throughput of triggered transactions (ToTT), throughput of pre-committed transactions (ToPT), and throughput of committed transactions (ToCT). They define the number of transactions that are triggered, pre-committed, and committed per time unit respectively.

$$ToTT : tri_count / (t_e - t_s)$$

$$ToPT : sch_count / (t_e - t_s)$$

$$ToCT : exe_count / (t_e - t_s)$$

Throughput reveals the performance of an algorithm in maintaining transaction concurrency. Higher level of concurrency achieved results in more transactions triggered, pre-committed, and committed concurrently, and shorter time ($t_e - t_s$) required to execute H ; as a result, higher throughput can be obtained.

Test Cases	CONFLICT_RESOLVING	PP_CONFLICT_STRATEGY	PC_CONFLICT_STRATEGY
Case 0	0	0	0
Case 1	1	0	0
Case 2	1	1	0
Case 3	1	1	1
Case 4	1	1	2
Case 5	1	2	0
Case 6	1	2	1
Case 7	1	2	2

Table 7.1: Simulation Test Cases

Schedule Productivity. It has three criteria: productivity of scheduling (PoS), productivity of executing (PoE), and productivity of transaction processing (PoTP). Productivity of scheduling is the ratio of pre-committed transactions to triggered transactions and it measures the performance of the scheduling algorithm (or *TransactionScheduler*). Productivity of executing is the ratio of committed transactions to pre-committed transactions and it measures the performance of the transaction execution algorithm (or *TransactionExecutor*). Productivity of transaction processing is the ratio of executed transactions to triggered transactions and it measures the overall performance of transaction processing algorithm (including scheduling and executing).

$$PoS : sch_count/tri_count$$

$$PoE : exe_count/sch_count$$

$$PoTP : exe_count/tri_count.$$

Productivity reveals the performance of an algorithm in resolving conflicts and exceptions. In case of potential transaction conflicts or exceptions, the better performance a handling (conflict resolving and exception handling) algorithm can achieve, the more transactions can be scheduled or executed, and the higher productivity ratio can be attained.

7.4 Simulation Cases

In each simulation group, we run eight test cases, as shown in Table 7.1.

- Case 0. Conflict resolving is turned off ($CONFLICT_RESOLVING=0$). This causes the *ResourceServer* to send an ACK message for all resource reservation requests. Since a REJ message from the server triggers conflict resolving, in this case, no conflict resolving will be applied by a CPS entity. However, even though the server sends ACK messages back every time, it still checks whether a reservation causes conflicts with previous reservations. Case 0 is used to show how many conflict reservations transactions will cause if they are scheduled.
- Case 1. Conflict resolving is turned on, but PP and PC strategies are not set. In this case, *ResourceServer* processes reservation requests normally and no *transaction preemption* is allowed ($PC_CONFLICT_STRATEGY=0$), and *TransactionScheduler* **directly aborts** a transaction if its resource reservation is rejected ($PP_CONFLICT_STRATEGY=0$). Case 1 is used to show the algorithm performance if direct abortion is applied to a conflicting transaction.
- Case 2. Conflict resolving is turned on, PP strategy is set to *Win-Lose* ($PP_CONFLICT_STRATEGY=1$), and PC strategy is not set. *ResourceServer* processes reservation requests normally without considering transaction preemption, while *TransactionScheduler* applies Win-Lose resolving strategy to a conflicting transaction: the transaction is aborted, but an **alternative** transaction will be created to replace it.
- Case 3. The same as Case 2 except that, with $PC_CONFLICT_STRATEGY=1$, *ResourceServer* now allows transaction preemption only on reservations with *Pending* state (Section 6.5.1).
- Case 4. The same as Case 3 except that *ResourceServer* allows transaction preemption on reservations with *Pending*, *Succeed*, or *PendingACK* states ($PC_CONFLICT_STRATEGY=2$).
- Case 5. The same as Case 2 except that PP strategy is set to *Win-Win* ($PP_CONFLICT_STRATEGY=2$). With Win-Win strategy, a conflicting transaction is first

adjusted (dynamic transaction adjustment in Section 5.3.5). If the adjustment succeeds, the adjusted transaction replaces the conflicting one; otherwise, an alternative transaction is created and the conflicting transaction is aborted.

- Case 6. The same as Case 3 except that PP strategy is set to *Win-Win*.
- Case 7. The same as Case 4 except that PP strategy is set to *Win-Win*.

For PC Conflict resolving strategy (specifically, transaction preemption), as discussed in Section 5.3.4, it is applied only when a transaction has a higher priority than another transaction, and its priority reaches the emergency level (**priPE**). In our implementation, **priPE** is set to 1. Thus, PC strategy is considered only when when a transaction's priority reaches 1 (refer to Section 6.3.3 for transaction priority settings). A transaction generated from the scenario file is granted with a priority of 4 . When it is adjusted, the adjusted transaction is granted with a priority of 3, and an exception-handling transaction has a priority of 2. However, in all simulation groups, we do not simulate emergency-handling transactions, and so no transaction preemption is applied. Thus, you are expected to see no difference caused by the transaction preemption strategy in all simulation results when only *PC_CONFLICT_STRATEGY* changes.

7.5 Group 0: Verifying The Simulation Platform

Group 0 is designed to evaluate the simulation platform. In this group, only one CPS entity is configured in the CPS Network, and 26 transactions are specified in its scenario file.

As shown in Figure 7.2, the CPS entity (a car) starts with system state $(2.0, 0.0, ROAD_1, 1, 0)$ (i.e., speed, acceleration, road, lane, and segment), and moves from lane 1 to 5 (four *ChangeToRightLaneTransactions*), and then from lane 5 to 1 (four *ChangeToLeftLaneTransactions*). Between each pair of changing lane transactions, it performs *ConstantSpeedTransaction*, *AccelerateTransaction*, and *DecelerateTransaction* randomly (Figure 7.3a).

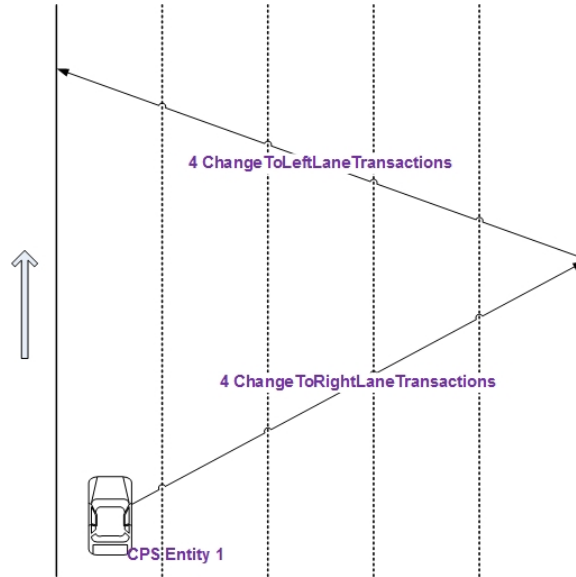


Figure 7.2: Simulation Group 0

Eight test cases all give the same simulation result as shown in Figure 7.3b. Two sample result files from *CPSNetwork* and *CyberPhysicalSystem* are available in Appendix A and B. The simulation result shows that:

- 26 transactions are generated (*GeneratedTranCount* in Figure 7.3b) by *TransactionGenerator* component.
- 26 transactions are scheduled (*ScheduledTranCount*) without any potential conflict by *TransactionScheduler* component.
- 26 transactions are executed completely by *TransactionExecutor* component (*ExecutedTranCount* and *CompletedTranCount*).
- 78 messages are sent through the network (*Number of Messages*).

Since there is only one CPS entity in the CPS Network, we expect that all its transactions are successfully scheduled and executed no matter whether conflict resolving function is enabled and which strategy we choose to resolve PP and PC conflict. The simulation results meet our expectation. Combined with the operation logs (Figure 7.4) of each component in

```

4
5 # Group 0
6 #ConstantSpeedTransaction,2000,0.0
7 #AccelerateTransaction,3000,4.0
8 #ChangeToRightLaneTransaction,2000,30
9 #AccelerateTransaction,3000,6.0
10 #ConstantSpeedTransaction,2000,0.0
11 #ChangeToRightLaneTransaction,2000,30
12 #DecelerateTransaction,3000,4.0
13 #ConstantSpeedTransaction,2000,0.0
14 #ChangeToRightLaneTransaction,2000,30
15 #AccelerateTransaction,3000,6.0
16 #ConstantSpeedTransaction,2000,0.0
17 #ChangeToRightLaneTransaction,2000,30
18 #DecelerateTransaction,3000,4.0
19 #ConstantSpeedTransaction,2000,0.0
20 #ChangeToLeftLaneTransaction,2000,30
21 #AccelerateTransaction,3000,6.0
22 #ConstantSpeedTransaction,2000,0.0
23 #ChangeToLeftLaneTransaction,2000,30
24 #DecelerateTransaction,3000,4.0
25 #ConstantSpeedTransaction,2000,0.0
26 #ChangeToLeftLaneTransaction,2000,30
27 #AccelerateTransaction,3000,6.0
28 #ConstantSpeedTransaction,2000,0.0
29 #ChangeToLeftLaneTransaction,2000,30
30 #DecelerateTransaction,3000,4.0
31 #ConstantSpeedTransaction,2000,0.0
32

```

(a) Scenario File

```

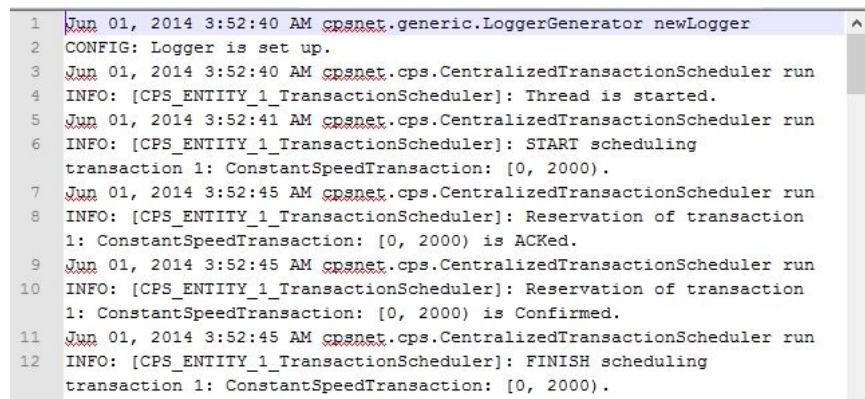
26 ##### STATISTICS
27 GeneratedTranCount=26
28 ConflictedOriginalTranCount=0
29 NotAdjustedTranCount=0
30 AdjustOnceTranCount=0
31 AdjustTwiceTranCount=0
32 AlternativeTranCount=0
33 AdjustSucceedTranCount=0
34 AdjustFailTranCount=0
35 AdjustOnceSucceedCount=0
36 AdjustOnceFailCount=0
37 AdjustTwiceSucceedCount=0
38 AdjustTwiceFailCount=0
39 AlterSucceedTranCount=0
40 AlterFailTranCount=0
41 ScheduledTranCount=26
42 ExecutedTranCount=26
43 CompleteTranCount=26
44 IncompleteTranCount=0
45 PreemptedTranCount=0
46 ConflictReservationCount=0
47
48
49 ##### MESSAGES
50 Number of Messages=78

```

(b) Simulation Results

Figure 7.3: Simulation Scenario and Result of Group 0

CyberPhysicalSystem, we conclude that these components operate as what they are designed for and carry out the transaction processing flow correctly.



```
1 Jun 01, 2014 3:52:40 AM cpsnet.generic.LoggerGenerator newLogger
2 CONFIG: Logger is set up.
3 Jun 01, 2014 3:52:40 AM cpsnet.cps.CentralizedTransactionScheduler run
4 INFO: [CPS_ENTITY_1_TransactionScheduler]: Thread is started.
5 Jun 01, 2014 3:52:41 AM cpsnet.cps.CentralizedTransactionScheduler run
6 INFO: [CPS_ENTITY_1_TransactionScheduler]: START scheduling
transaction 1: ConstantSpeedTransaction: [0, 2000).
7 Jun 01, 2014 3:52:45 AM cpsnet.cps.CentralizedTransactionScheduler run
8 INFO: [CPS_ENTITY_1_TransactionScheduler]: Reservation of transaction
1: ConstantSpeedTransaction: [0, 2000) is ACKed.
9 Jun 01, 2014 3:52:45 AM cpsnet.cps.CentralizedTransactionScheduler run
10 INFO: [CPS_ENTITY_1_TransactionScheduler]: Reservation of transaction
1: ConstantSpeedTransaction: [0, 2000) is Confirmed.
11 Jun 01, 2014 3:52:45 AM cpsnet.cps.CentralizedTransactionScheduler run
12 INFO: [CPS_ENTITY_1_TransactionScheduler]: FINISH scheduling
transaction 1: ConstantSpeedTransaction: [0, 2000).
```

Figure 7.4: Operation Log of TransactionScheduler

To successfully reserve resources for each transaction, three messages are sent: a REQ message from client to server, an ACK message from server to client, and a CON message from client to server. From the result, we know that 78 messages are sent through the communication network, which is exactly $3 * 26$. Checking the operation logs of *Network* and its sub-components (*NetworkMessageCollector* and *NetworkMessageDistributor*), we know that they work correctly regarding message delivery. Besides, the number of messages also proves that *TransactionScheduler*, and *ResourceServer* and its *ReservationWorker* work as expected with respect to sending requests and responses.

The total simulation time (not shown in Figure 7.3b) taken is 74,344 ticks (of the simulated global clock), and it is reasonable to observe that it is larger than the total execution time of those 26 transactions (Figure 7.3a), which is 61,000 ticks. Considering time costs for *CPSNetwork* to initialize and start a simulation, for each component of a CPS entity to process transactions, and for statistical information to be collected, the simulation time is in our expected range.

7.6 Group 1-5: Verifying The Conflict Resolving Algorithms

In this section, we present five simulation groups to show how conflict resolving algorithms are used to resolve PP and PC conflict in the pre-commit phase.

In the pre-commit phase, potential transaction conflicts are detected by *TransactionScheduler* and resolved by *ConflictResolver*. When a REJ message is received from the *ResourceServer* for the resource reservation of a transaction, *TransactionScheduler* knows that the transaction is in potential conflicts with others, and the conflicting resource usages are returned in the message body. Then, depending on whether conflict resolving function is enabled, the type of a conflict and a transaction, and which strategy is selected to resolve the conflict, *TransactionScheduler* calls the corresponding method of *ConflictResolver* to resolve conflicts. In the current implementation, *ConflictResolver* provides two types of methods (Figure 7.5):

- Methods that create alternative transactions. These methods start with “alter”.
- Methods that perform transaction adjustment. These methods start with “adjust”.

In each of these five groups, two CPS entities are simulated.

- Entity 1 is the reference entity, and Entity 2 is the entity we are testing.
- Entity 1 has only one *ConstantSpeedTransaction* to execute in all groups. However, Entity 2 executes one or two transactions, one of which is different in each group. Transactions of Entity 2 are designed in this way in order to test how conflicts are resolved for different types of transactions.
- Entity 1 has an earlier start time than Entity 2, and this makes sure that the transaction of Entity 1 always gets required resources reserved before transactions of Entity 2.

- `alternativeTransaction(Transaction, List<ResourceUsage>) : Transaction`
- `resolveConflicts(Transaction, List<ResourceUsage>) : Transaction`
- `adjustConstantSpeedTransaction(Transaction, List<ResourceUsage>) : Transaction`
- `adjustAccelerateTransaction(Transaction, List<ResourceUsage>) : Transaction`
- `adjustDecelerateTransaction(Transaction, List<ResourceUsage>) : Transaction`
- `adjustChangeToLeftLaneTransaction(Transaction, List<ResourceUsage>) : Transaction`
- `adjustChangeToRightLaneTransaction(Transaction, List<ResourceUsage>) : Transaction`
- `alterConstantSpeedTransaction(Transaction, List<ResourceUsage>) : Transaction`
- `alterAccelerateTransaction(Transaction, List<ResourceUsage>) : Transaction`
- `alterDecelerateTransaction(Transaction, List<ResourceUsage>) : Transaction`
- `alterChangeToLeftLaneTransaction(Transaction, List<ResourceUsage>) : Transaction`
- `alterChangeToRightLaneTransaction(Transaction, List<ResourceUsage>) : Transaction`
- `actionInConflict(Transaction, List<ResourceUsage>) : List<Pair<Integer, Action>>`
- `getAdjustedOnce() : int`
- `getAdjustedTwice() : int`
- `getAlternative() : int`

Figure 7.5: Conflict Resolver

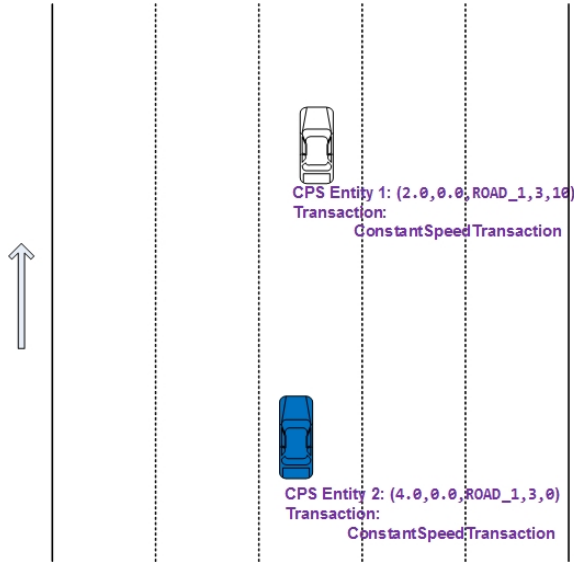
- To ensure conflicts are indeed created for testing, besides the start time, start system states of both Entity 1 and 2 are tuned purposely to create desired PP or PC conflict.

7.6.1 Group 1: ConstantSpeedTransaction

The scenario for Group 1 is shown in Figure 7.6a. Both entities execute a single *ConstantSpeedTransaction*, which is composed of a single *ConstantSpeedAction*.

The start system state of Entity 1 is $(2.0, 0.0, ROAD_1, 3, 10)$, and that of Entity 2 is $(4.0, 0.0, ROAD_1, 3, 0)$. So Entity 1 is ahead of Entity 2 by 10 meters in lane 3, but Entity 2 has a higher speed. The *startTime* of Entity 1 is set to 0, and the time for Entity 2 is 1000 (ticks). When the simulation starts, Entity 1 gets its reservation request for its transaction granted first by the *ResourceServer*, and Entity 2's transaction is in conflict with Entity 1's.

For Entity 1, based on our design, its transaction gets scheduled and executed successfully in all test cases. However, for Entity 2, different test cases may give different results (Figure 7.6b).



(a) Group 1 - Scenario

CPS Entity 2	Case 0	Case 1	Case 2, 3, 4	Case 5, 6, 7
GeneratedTranCount	1	1	1	1
ConflictedOriginalTranCount	1	1	1	1
NotAdjustedTranCount		1	1	
AdjustOnceTranCount				1
AdjustTwiceTranCount				1
AlternativeTranCount			1	
AdjustSucceedTranCount				1
AdjustFailTranCount				1
AdjustOnceSucceedCount				
AdjustOnceFailCount				1
AdjustTwiceSucceedCount				1
AdjustTwiceFailCount				
AlterSucceedTranCount			1	
AlterFailTranCount				
ScheduledTranCount	1		1	1
ExecutedTranCount	1		1	1
CompleteTranCount	1		1	1
IncompleteTranCount				
PreemptedTranCount				
ConflictReservationCount	1	1	1	2
NumberOfMessages	6	6	9	12

(b) Group 1 - Result

Figure 7.6: Simulation Group 1

- In Case 0, when conflict resolving is disabled, *ConflictReservationCount* indicates that there is one resource reservation (Entity 2's) conflicting with the existing reservation (Entity 1's). If both entities execute their transactions, a real-time conflict will occur and the execution of both transactions will fail.
- In Case 1, when a REJ message from *ResourceServer* is received, the conflicting transaction is directly aborted by Entity 2, which causes that no transaction is ever executed by Entity 2.
- Case 2, 3, and 4 have the same simulation result because of the non-applicable PC strategy (discussed in Section 7.4). In these cases, Win-Lose strategy is applied by Entity 2. Thus, we see that the conflicting transaction is not adjusted, but one alternative transaction is created to replace it. The alternative transaction has no conflict with Entity 1's transaction (*AlterSucceedTranCount*=1), and gets scheduled and executed successfully. The *alterConstantSpeedTransaction* method of *ConflictResolver* is used to create the alternative transaction. In default, the alternative transaction for

ConstantSpeedTransaction is *ChangeToLeftLaneTransaction* or *ChangeToRightLaneTransaction*.

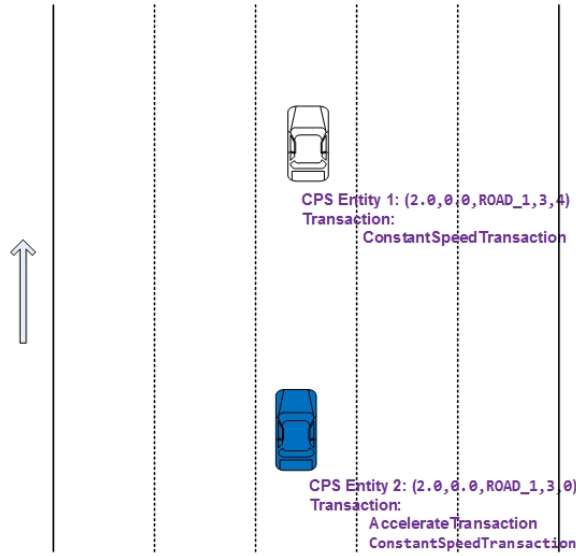
- Case 5, 6, and 7 also have the same result. In these cases, Win-Win strategy is applied. So the conflicting transaction of Entity 2 is adjusted first, and if the adjustment does not resolve conflicts, an alternative transaction is created. The *adjustConstantSpeedTransaction* method of *ConflictResolver* tries adjusting a conflicting transaction at most two times. First, it inserts an *IncreaseSpeedAction* at the beginning of the action sequence of the transaction. If it fails, a *DecreaseSpeedAction* is tried instead. Both adjustments are **sequence adjustment** (Section 5.3.5). The simulation result shows that the first adjustment doesn't work (*AdjustOnceFailCount=1*), but the second does (*AdjustTwiceSucceedCount=1*).

From the above analysis, we know that the application of Win-Lose and Win-Win strategy resolves the potential transaction conflict and increases the number of transactions that are executed. In Case 2-7, two transactions (from both Entity 1 and Entity 2) are executed successfully, compared to 0 transaction being executed if no conflict resolving is applied (Case 0). However, the number of messages transmitted in the network also increases because the more adjustments are performed, the more reservation requests are sent out. As mentioned earlier, a successful reservation takes three messages. An unsuccessful reservation also takes three messages: a REQ message from client to server, a REJ message from server to client, and a CAN message from client to server.

7.6.2 Group 2: AccelerateTransaction

In this group, we show how a conflicting *AccelerateTransaction* is adjusted to resolve conflicts. An *AccelerateTransaction* is composed of a single *IncreaseSpeedAction*.

Figure 7.7 shows the simulation scenario and result of Group 2. Entity 1 is set up as a reference just like Group 1, and Entity 2 has two transactions: *AccelerateTransaction* and *ConstantSpeedTransaction*. The start system states of both Entity 1 and 2 (Figure 7.7a)



(a) Group 2 - Scenario

CPS Entity 2	Case 0	Case 1	Case 2, 3, 4	Case 5, 6, 7
GeneratedTranCount	2	2	2	2
ConflictedOriginalTranCount		1	1	1
NotAdjustedTranCount		1	1	
AdjustOnceTranCount				1
AdjustTwiceTranCount				1
AlternativeTranCount			1	1
AdjustSucceedTranCount				
AdjustFailTranCount				2
AdjustOnceSucceedCount				
AdjustOnceFailCount				1
AdjustTwiceSucceedCount				
AdjustTwiceFailCount				1
AlterSucceedTranCount			1	1
AlterFailTranCount				
ScheduledTranCount	2	1	2	2
ExecutedTranCount	2	1	2	2
CompleteTranCount	2	1	2	2
IncompleteTranCount				
PreemptedTranCount				
ConflictReservationCount	1	1	1	3
NumberOfMessages	9	9	12	18

(b) Group 2 - Result

Figure 7.7: Simulation Group 2

are specially tuned to make *AccelerateTransaction* in conflict with the *ConstantSpeedTransaction* of Entity 1.

The simulation result is shown in Figure 7.7b.

- When conflict resolving is disabled, the *ResourceServer* finds one reservation (Case 0, *ConflictReservationCount*=1) in conflict with previous ones (*AccelerateTransaction* of Entity 2).
- When direct abortion of a conflicting transaction is applied (Case 1), the *AccelerateTransaction* is aborted, and only the *ConstantSpeedTransaction* transaction of Entity 2 gets executed finally.
- If Win-Lose strategy is used, the conflicting *AccelerateTransaction* is replaced with an alternative *ChangeToLeftLaneTransaction* or *ChangeToRightLaneTransaction*. As the result of Case 2, 3, and 4 shows, the alternative transaction resolves the conflict.
- If Win-Win strategy is used, the *AccelerateTransaction* is adjusted two times. How-

ever, both adjustment fail to resolve the conflict (*AdjustFailTranCount=2*), and finally the alternative transaction resolves the conflict and gets executed (*AlterSucceedTranCount=1*).

The adjustment methods (*adjustAccelerateTransaction*) defined in *ConflictResolver* for *AccelerateTransaction* performs **action adjustment** (Section 5.3.5) by changing either the *targetSpeed* or *endTime* of the *IncreaseSpeedAction* in the action sequence. In the first try, *targetSpeed* is reduced by 1, or *endTime* is decreased or increased by 1000 ticks, depending on whether the adjustment will cause an invalid system state regarding the adjustment. In the second try, *targetSpeed* is increased by 1. When two adjustment attempts fail to resolve the conflict, an alternative transaction is created instead.

More adjustment tries cause more messages to be transmitted. In Case 4, 5, 6, six reservations requests are sent to reserve resources for six transactions (the original 3 transactions defined in the scenario file, plus two adjusted transaction, and one alternative transaction), which creates a total of 18 messages.

7.6.3 Group 3: DecelerateTransaction

In this group, we show how to resolve the potential conflict of a *DecelerateTransaction*.

As the scenario in Figure 7.8a shows, Entity 2 is ahead of Entity 1 by 8 meters, and it executes two transactions: *DecelerateTransaction* (consisting of a single *DecreaseSpeedAction*) and *ConstantSpeedTransaction*. Although they have the same start speed, the *DecelerateTransaction* reduces the speed of Entity 2, and causes a conflict with Entity 1's transaction. Because the speed is reduced, *ConstantSpeedTransaction* is also in conflict with Entity 1's (Case 0, *ConflictReservationCount=2*). However, if *DecelerateTransaction* is aborted (Case 1), the speed of Entity 2 stays the same as Entity 1's, and the *ConstantSpeedTransaction* of Entity 2 then will not conflict with Entity 1. If Win-Lose strategy is used (Case 2, 3, and 4), *DecelerateTransaction* is replaced with an alternative *ChangeToRightLaneTransaction* or *ChangeToLeftLaneTransaction*, which resolves the conflicts, and makes

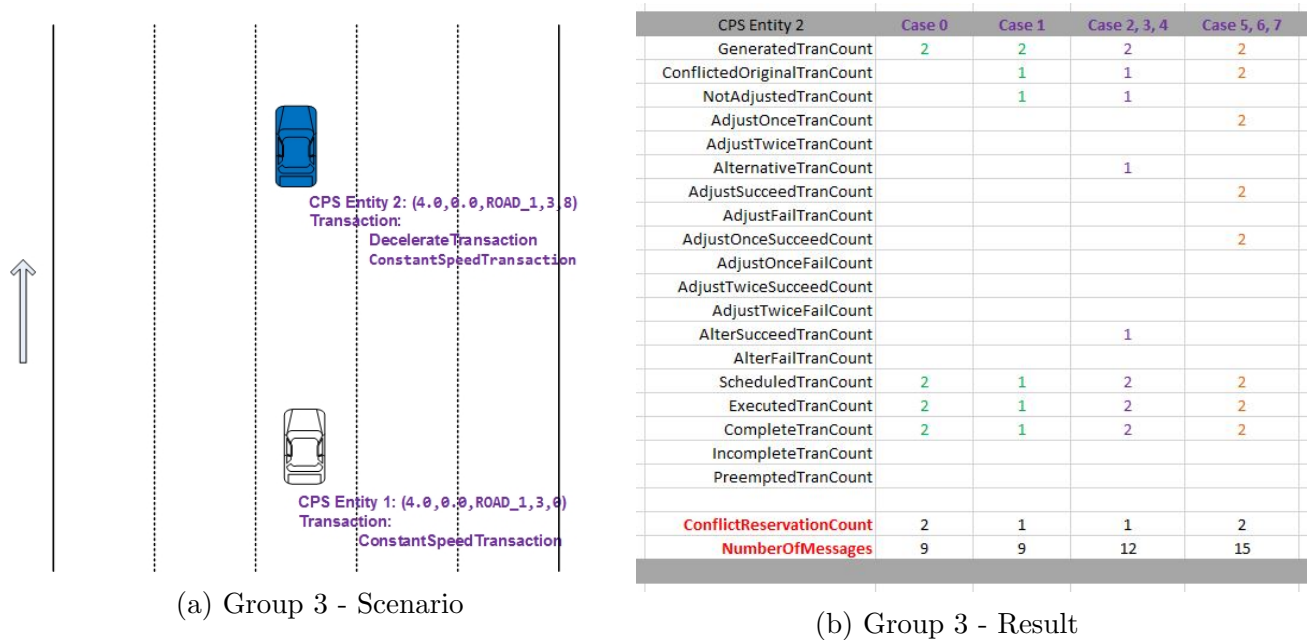
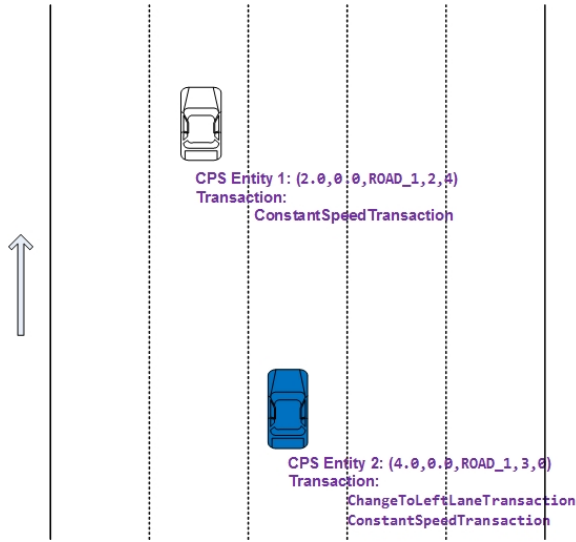


Figure 7.8: Simulation Group 3

the alternative transaction itself and the following *ConstantSpeedTransaction* conflict-free.

In Case 5, 6, and 7, the *DecelerateTransaction* is adjusted once and succeeds to avoid conflict with Entity 1's. What the original *DecelerateTransaction* does is to reduce the speed of Entity 2 from 4.0 to 2.0. After the action adjustment, the *targetSpeed* is increased by 1, i.e., the adjusted *DecelerateTransaction* now reduces the speed from 4.0 to 3.0. However, the following *ConstantSpeedTransaction* is still in conflict with Entity 1's, and a sequence adjustment is performed to insert an *IncreaseSpeedAction* in the beginning of its action sequence. This temporary action brings the speed back to 4.0, and makes the adjusted transaction conflict-free. Thus, we see two adjusted transactions and both succeed to resolve conflicts in Figure 7.8b (*AdjustOnceSucceedCount*=2). In these three cases, five transactions (including the adjusted ones) go through the scheduling process, and result in a total of 15 messages in transmission.



(a) Group 4 - Scenario

CPS Entity 2	Case 0	Case 1	Case 2, 3, 4	Case 5, 6, 7
GeneratedTranCount	2	2	2	2
ConflictedOriginalTranCount		1	1	1
NotAdjustedTranCount		1	1	
AdjustOnceTranCount				1
AdjustTwiceTranCount				
AlternativeTranCount			1	
AdjustSucceedTranCount				1
AdjustFailTranCount				
AdjustOnceSucceedCount				1
AdjustOnceFailCount				
AdjustTwiceSucceedCount				
AdjustTwiceFailCount				
AlterSucceedTranCount			1	
AlterFailTranCount				
ScheduledTranCount	2	1	2	2
ExecutedTranCount	2	1	2	2
CompleteTranCount	2	1	2	2
IncompleteTranCount				
PreemptedTranCount				
ConflictReservationCount	1	1	1	1
NumberOfMessages	9	9	12	12

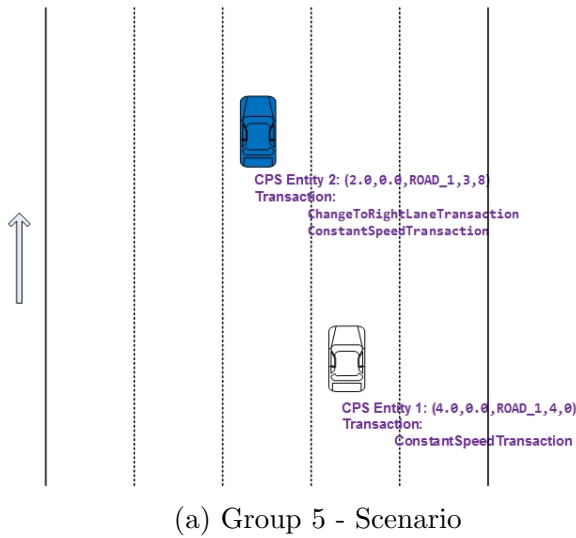
(b) Group 4 - Result

Figure 7.9: Simulation Group 4

7.6.4 Group 4: ChangeToLeftLaneTransaction

In this group, we show how a conflicting *ChangeToLeftLaneTransaction* is handled. A *ChangeToLeftLaneTransaction* consists of three actions in sequence: *ConstantSpeedAction*, *TurnLeftAction*, and *ConstantSpeedAction*. The alternative transaction *ConflictResolver* defines for a *ChangeToLeftLaneTransaction* is a *ConstantSpeedTransaction*. When it is being adjusted, an *IncreaseSpeedAction* is prepended to the action sequence in the first adjustment attempt. In the second attempt, a *DecreaseSpeedAction* is used instead. Figure 7.9 shows the simulation scenario and result.

In this given scenario, the third action of *ChangeToLeftLaneTransaction* is in conflict with Entity 1's *ConstantSpeedTransaction* after it changes to the 2nd lane. If conflict resolving is disabled, we can see that the *ResourceServer* detects the conflict (*ConflictReservationCount*=1). If the conflicting *ChangeToLeftLaneTransaction* is aborted (Case 1), Entity 2 executes the remaining *ConstantSpeedTransaction*, which has no conflict with Entity 1's (*ExecutedTranCount*=1). In Case 2, 3 and 4, Win-Lose strategy creates an



CPS Entity 2	Case 0	Case 1	Case 2, 3, 4	Case 5, 6, 7
GeneratedTranCount	2	2	2	2
ConflictedOriginalTranCount		1	1	1
NotAdjustedTranCount		1	1	
AdjustOnceTranCount				1
AdjustTwiceTranCount				1
AlternativeTranCount			1	
AdjustSucceedTranCount				1
AdjustFailTranCount				1
AdjustOnceSucceedCount				
AdjustOnceFailCount				1
AdjustTwiceSucceedCount				1
AdjustTwiceFailCount				
AlterSucceedTranCount			1	
AlterFailTranCount				
ScheduledTranCount	2	1	2	2
ExecutedTranCount	2	1	2	2
CompleteTranCount	2	1	2	2
IncompleteTranCount				
PreemptedTranCount				
ConflictReservationCount	1	1	1	2
NumberOfMessages	9	9	12	15

(b) Group 5 - Result

Figure 7.10: Simulation Group 5

alternative *ConstantSpeedTransaction* to replace the conflicting *ChangeToLeftLaneTransaction* and the conflict is resolved successfully. When Win-Win strategy is used, however, it succeeds to keep the *ChangeToLeftLaneTransaction* by prepending an *IncreaseSpeedAction* to its action sequence, which avoids the potential conflict: *AdjustOnceTranCount*=1 and *AdjustOnceSucceedCount*=1. The number of messages grows linearly with the number of transactions that send out reservation requests: 4 transactions and 12 messages.

7.6.5 Group 5: ChangeToRightLaneTransaction

In this group, we show how a conflicting *ChangeToRightLaneTransaction* is handled. Similar to *ChangeToLeftLaneTransaction*, *ChangeToRightLaneTransaction* also consists of three actions in sequence: *ConstantSpeedAction*, *TurnRightAction*, and *ConstantSpeedAction*. Besides, the same measures are taken for *ConflictResolver* to create an alternative transaction and to adjust the transaction.

The simulation scenario and result is shown in Figure 7.10. Similar to Group 4, the third action of the *ChangeToRightLaneTransaction* has a conflict with Entity 1's *ConstantSpeed*

Transaction, and both groups have the same simulation results except Case 5, 6, and 7.

In Case 5, 6, and 7, it takes two adjustment attempts for *ConflictResolver* to resolve the potential conflict: *AdjustOnceFailCount=1* and *AdjustTwiceSucceedCount=1*. The first attempt prepends an *IncreaseSpeedAction* (increasing speed by 1) to the action sequence, but Entity 1's still has a higher speed and will catch up with Entity 2 after it changes lane. The second attempt prepends a *DecreaseSpeedAction* (decreasing speed by 1), which makes Entity 1 move pass Entity 2 before Entity 2 changes lane, and the conflict is resolved. Since one more attempt is tried to resolve conflicts, one more transaction is generated than Group 4 and this results in a total of 15 messages being transmitted.

7.6.6 Discussion

In the above five simulation groups, we show how potential conflicts of different types of transactions are resolved using different conflict resolving strategies. Simulation results show that, with conflict resolving enabled, the number of **executed** transactions is more than that when conflict resolving is not applied. However, conflict resolving doesn't come without cost. For example, more time is required to carry the adjustment attempts, and more messages are transmitted.

One thing to note is that algorithms we design in those methods of *ConflictResolver* to create alternative transactions and to adjust conflicting transactions are not necessarily the best solution. We design these algorithms by considering only general situations that may cause conflicts in a type of transactions, and they can be improved in different ways. For example, by analyzing the characteristics of the conflicting resource usages and locating the conflicting action in the transaction, one can make scenario-specific adjustment (either action or sequence) to the transaction and improve the probability of success of the first adjustment attempt, which in turn can reduce the number of messages being transmitted and save time for *TransactionScheduler* to schedule transactions. Moreover, machine learning techniques [59] can be used to learn the patterns of different types of transaction conflicts

and provide a better prediction on how to resolve conflicts.

7.7 Group 6: A Comprehensive Simulation

So far, we have gone through six simulation groups and shown the performance of the simulation platform and the transaction model in different aspects. In this group, we perform a comprehensive simulation that puts all aspects together. It includes 10 CPS entities, and each entity executes 12 to 26 transactions (Figure 7.11). Together these 10 entities forms a schedule with 188 transactions. By simulating the schedule, we can have a more comprehensive view of the simulation platform and the transaction processing algorithms. Moreover, we test the scalability of the simulation platform by increasing the number of CPS entities and transactions separately and check the performance of the simulation platform.

Simulation results show that our simulation platform behaves as expected even with a large load of CPS entities and transactions, and that our transaction processing algorithms are able to improve the throughput and productivity of a schedule comparing with algorithms when no conflict resolving is applied or direct abortion of a conflicting transaction is used.

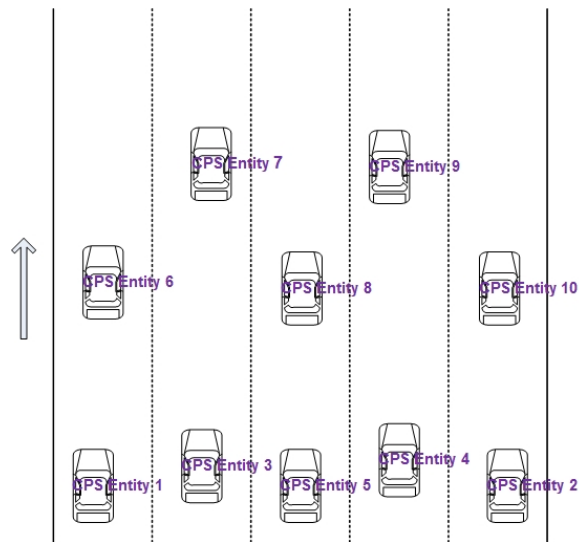


Figure 7.11: Group 6 - Scenario

7.7.1 Simulation Scenario

The ten CPS entities execute different sequences of transactions that make them move on the road in different patterns. Their configurations, including their start system states, are listed in Figure 7.12. Their relative positions in the road is illustrated by Figure 7.11.

```
31 # Group 6
32 CPS_ENTITY_1;2.0,0.0,ROAD_1,1,0,null;0;100000;CPS_ENTITY_1_SCENARIO.scenario
33 CPS_ENTITY_2;2.0,0.0,ROAD_1,5,0,null;0;100000;CPS_ENTITY_2_SCENARIO.scenario
34 CPS_ENTITY_3;3.0,0.0,ROAD_1,2,0,null;0;100000;CPS_ENTITY_3_SCENARIO.scenario
35 CPS_ENTITY_4;3.0,0.0,ROAD_1,4,0,null;0;100000;CPS_ENTITY_4_SCENARIO.scenario
36 CPS_ENTITY_5;3.0,0.0,ROAD_1,3,0,null;0;100000;CPS_ENTITY_5_SCENARIO.scenario
37 CPS_ENTITY_6;2.0,0.0,ROAD_1,1,5,null;0;100000;CPS_ENTITY_6_SCENARIO.scenario
38 CPS_ENTITY_7;4.0,0.0,ROAD_1,2,8,null;0;100000;CPS_ENTITY_7_SCENARIO.scenario
39 CPS_ENTITY_8;2.0,0.0,ROAD_1,3,5,null;0;100000;CPS_ENTITY_8_SCENARIO.scenario
40 CPS_ENTITY_9;4.0,0.0,ROAD_1,4,8,null;0;100000;CPS_ENTITY_9_SCENARIO.scenario
41 CPS_ENTITY_10;2.0,0.0,ROAD_1,5,5,null;0;100000;CPS_ENTITY_10_SCENARIO.scenario
42
```

Figure 7.12: Group 6 - CPS Configuration

The scenario file of each CPS entity specifies a sequence of transactions to be generated, scheduled, and executed. These transactions lead each CPS entity to a different moving pattern on the road.

- CPS entity 1 starts at the 1st lane and keeps changing lanes from lane 1 to lane 5, and then from lane 5 back to lane 1. Between two changing lane transactions, it accelerates, decelerates, or moves at constant speed. 26 transactions are specified in its scenario file. CPS entity 2 also executes 26 transactions. However, on the contrary, it starts at lane 5 and changes lanes until lane 1, and then from lane 1 back to lane 5.
- CPS entity 3 starts at lane 2 and also keeps changing lanes. However, the lanes it change to are restricted to lane 1, 2, and 3. CPS entity 4 starts at lane 4 and does the opposite. It keeps changing lanes within lane 3, 4, and 5. Between changing lane transactions, just like entity 1 and 2, Entity 3 and 4 randomly accelerate, decelerate, or move at constant speed.

- CPS entity 5 starts at lane 3 and its movement is similar to entity 1-4, but the lanes are restricted to lane 2, 3, and 4.
- For CPS entities 6, 8, and 10, each executes 12 transactions, and their movements are within a single lane without any changing lane transactions. They may accelerate, decelerate, or move at constant speed.
- For CPS entities 7 and 9, they both executes 17 transactions, and their movements are also restricted to a single lane.

Unlike Group 1-5, CPS entities in this group have the same start time 0. This means there is no specific order to be followed when they schedule and execute their transactions. Each CPS entity will compete with another for resource usages of their transactions, while the *ResourceServer* processes the reservation requests in a FCFS (First Come, First Served) order.

Although scenario files specify the sequence of transactions to be execute by a CPS entity, as what we have seen in previous simulation groups, actual transactions scheduled and executed by a CPS entity are subject to change if conflict resolving is applied, e.g., one transaction is in conflict and is replaced by an alternative or an adjusted transaction.

7.7.2 Simulation Result

Figure 7.13 show a sample simulation result collected through the *CPSNetwork*, and an average result of 10 simulation runs is shown in Figure 7.14.

In all test cases, we notice that the numbers of *ScheduledTranCount*, *ExecutedTranCount*, and *CompleteTranCount* are equal. This is because, in our current implementation, we assume that the commit-phase always works as expected and a scheduled transaction always gets executed successfully without exceptions. In the following discussion, we refer to Figure 7.13 and Figure 7.14 at the same time.

CPSNetwork	Case 0	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7
GeneratedTranCount	188	188	188	188	188	188	188	188
ConflictedOriginalTranCount		18	22	15	17	20	20	14
NotAdjustedTranCount		18	37	29	33			
AdjustOnceTranCount						20	20	14
AdjustTwiceTranCount						10	13	7
AlternativeTranCount			23	16	18	5	10	5
AdjustSucceedTranCount						15	10	9
AdjustFailTranCount						15	23	12
AdjustOnceSucceedCount						10	7	7
AdjustOnceFailCount						10	13	7
AdjustTwiceSucceedCount						5	3	2
AdjustTwiceFailCount						5	10	5
AlterSucceedTranCount			8	2	2		1	1
AlterFailTranCount			15	14	16	5	9	4
ScheduledTranCount	188	170	174	175	173	183	179	184
ExecutedTranCount	188	170	174	175	173	183	179	184
CompleteTranCount	188	170	174	175	173	183	179	184
IncompleteTranCount								
PreemptedTranCount								
ConflictReservationCount	25	18	40	30	37	42	59	32
NumberOfMessages	564	564	633	612	618	669	693	642

Figure 7.13: Group 6 - Simulation Result of CPS Network

	CPSNetwork	Case 0	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	2, 3, 4	5, 6, 7
193	GeneratedTranCount	188	188	188	188	188	188	188	188	188	188
195	ConflictedOriginalTranCount	0	17.9	17.3	15.3	17.5	16.2	17.8	18.6	16.7	17.53333
196	NotAdjustedTranCount	0	17.9	31.4	28.7	32	0	0	0	30.7	0
197	AdjustOnceTranCount	0	0	0	0	0	16.2	17.8	18.6	0	17.53333
198	AdjustTwiceTranCount	0	0	0	0	0	8.2	10.4	10.1	0	9.56667
199	AlternativeTranCount	0	0	18.7	16.8	19	5.4	8.1	7.4	18.16667	6.96667
200	AdjustSucceedTranCount	0	0	0	0	0	10.8	9.7	11.2	0	10.56667
201	AdjustFailTranCount	0	0	0	0	0	13.6	18.5	17.5	0	16.53333
202	AdjustOnceSucceedCount	0	0	0	0	0	8	7.4	8.5	0	7.96667
203	AdjustOnceFailCount	0	0	0	0	0	8.2	10.4	10.1	0	9.56667
204	AdjustTwiceSucceedCount	0	0	0	0	0	2.8	2.3	2.7	0	2.6
205	AdjustTwiceFailCount	0	0	0	0	0	5.4	8.1	7.4	0	6.96667
206	AlterSucceedTranCount	0	0	4.3	3.4	4.2	0.7	1.1	1	3.96667	0.93333
207	AlterFailTranCount	0	0	14.4	13.4	14.8	4.7	7	6.4	14.2	6.03333
208	ScheduledTranCount	188	170.1	175.3	176.1	175	183.3	181	181.6	175.4667	181.9667
209	ExecutedTranCount	188	170.1	175.3	176.1	175	183.3	181	181.6	175.4667	181.9667
210	CompleteTranCount	188	170.1	175.3	176.1	175	183.3	181	181.6	175.4667	181.9667
211	IncompleteTranCount	0	0	0	0	0	0	0	0	0	0
212	PreemptedTranCount	0	0	0	0	0	0	0	0	0	0
213	ConflictReservationCount	25	18	33.8	31.4	35	36	51.3	47.3	33.4	44.86667
214	NumberOfMessages	564	564	620.1	614.4	621	642.9	672.9	672.3	618.5	662.7
215	EndTime (ticks)	79345	73943.2	74653	73943.6	74110.5	76870.7	76291.7	76910	74235.7	76690.8

Figure 7.14: Group 6 - Simulation Result of 10 Runs

Case 0

When conflict resolving is not applied, among the 188 transactions, 25 transactions are found to have conflicting resource usages with some transaction whose expected execution time is before theirs (Case 0, *ConflictReservationCount*=25).

Since the *ResourceServer* only counts the number of conflicting reservations, the result for *ScheduledTranCount*, *ExecutedTranCount*, and *CompleteTranCount* are not useful for our analysis. Currently, we don't have a way to measure how many transactions will be scheduled and executed when no conflict resolving is used. In the real world, taking a running car as an example, when transaction conflicts occur, it always indicates car collisions, and the car is most likely not able to execute any more transactions. So we can expect that the number of transactions failed to be executed is way more than the number of conflicting reservations detected by the *ResourceServer*.

Another thing to note is, compared with other cases, *ConflictReservationCount* of Case 0 is stable and stays 25 in all our simulation runs. We will explain the reason later.

Case 1

In Case 1, when a conflicting transaction is directly aborted without alternative transactions, 18 transactions are aborted because of conflicting reservations, and 170 transactions are executed. We can see that the number of aborted transactions is smaller than 25. This is because, in any CPS entity, when one of its transaction is aborted, the next transaction specified in the scenario file replaces the aborted one. This change affects the resource usages of all following transactions since their start system states are changed, which further changes the number of conflicting reservations detected in the *ResourceServer*.

However, in all our simulation runs, the number of *ConflictReservationCount* and aborted transactions is not fixed and subject to randomness to some extent. It is caused by the randomness of the scheduling process. Assume that two CPS entities (E_1 and E_2) have two conflicting transactions (T_1 and T_2), and both transactions don't have any conflict with

previous transactions that successfully make their reservations. The order that their reservation requests arrive at the *ResourceServer* affects the number of conflicting reservations for transactions following them in the schedule. If T_1 's request arrives first, T_2 will be detected by the server to have a conflict with T_1 , and will be aborted by E_2 after receiving a REJ message from the server. If T_2 's request arrives first, then T_1 will be aborted by E_1 . We know that the abortion of a transaction will affect all following transactions of the same CPS entity by changing their start system states. Thus, the two different orders actually create two different schedules, which have different number of conflicting reservations. Similar situations exist in other cases, as we will see later.

Case 2, 3, and 4

When Win-Lose strategy is applied to abort a conflicting transaction and replace it with an alternative one in Case 2, 3, and 4, we see a rise in the number of scheduled and executed transactions compared with Case 1. However, more conflict reservations are detected by the *ResourceServer*, and more messages are sent and received. This is because, every time a transaction is aborted, an alternative one is created and scheduled. Thus, the *Transaction Scheduler* of a CPS entity tries to reserve resources for transactions more than the number specified in the scenario file. With the increase of scheduling tries, more conflicts and messages will be caused, and the simulation time taken for a simulation also rises. However, despite of these rising costs, more transactions are successfully scheduled and executed than Case 1.

Unlike what we have observed in Group 1-5, where these three cases always have the same simulation result, the simulation results are different in Group 6. This is because, in Group 1-5, we specify the start times of CPS entities in order to create the conflict situations we need, and there is only two CPS entities and 2-3 transactions in each group. But in Group 6, no specific order is specified between CPS entities. Due to the randomness of the scheduling process, different cases and different simulation runs of a case produce different schedules and results. However, the difference has nothing to do with PC conflict

resolving strategy. As we mention in Section 7.4, no transaction preemption will be applied by the *ResourceServer* will in all simulation groups.

Case 5, 6, and 7

When Win-Win strategy is applied to make adjustment to a conflicting transaction, we see more number of transactions are scheduled and executed in Case 5-7 compared with Case 2-4. This is because Win-Win strategy (using transaction adjustment) has a better performance in resolving transaction conflicts (discussed later). However, due to more scheduling attempts for adjusted and alternative transactions, we see an increase in the number of transmitted messages and conflict reservations compared with Case 2-4. Moreover, the average time taken for a simulation increases compared with that of Case 2-4.

Similar to Case 2-4, due to the randomness of the scheduling process, randomness is also observed in the simulation results of Case 5-7, either the results for Case 5, 6 and 7 are different, or the results for different simulation runs of the same case are different.

7.7.3 Statistics

In Section 7.3, we propose several criteria to evaluate the performance of our transaction processing and conflict resolving algorithms. These criteria focus on the throughput and productivity of a schedule. For our simulation, they are calculated in following ways.

$$ToTT = \text{GeneratedTranCount} / (\text{endTime} - \text{startTime})$$

$$ToPT = \text{ScheduledTranCount} / (\text{endTime} - \text{startTime})$$

$$ToCT = \text{ExecutedTranCount} / (\text{endTime} - \text{startTime})$$

$$PoS = \text{ScheduledTranCount} / \text{GeneratedTranCount}$$

$$PoE = \text{ExecutedTranCount} / \text{ScheduledTranCount}$$

$$PoTP = \text{ExecutedTranCount} / \text{GeneratedTranCount}$$

Besides above criteria, we also define the following criteria to evaluate the performance of methods of *ConflictResolver*: Successful Once-Adjustment Rate (SOAR), Successful Twice-

Adjustment Rate (STAR), Successful Adjustment Rate (SADR), Successful Alternative Rate (SALR), Successful Resolving Rate (SRR), Adjustment or Alternative Rate (AoAR), and Conflict Reservation Rate (CRR)

$$SOAR = AdjustOnceSucceedCount / AdjustOnceTranCount$$

$$STAR = AdjustTwiceSucceedCount / AdjustTwiceTranCount$$

$$SADR = AdjustSucceedTranCount / (AdjustOnceTranCount + AdjustTwiceTranCount)$$

$$SALR = AlterSucceedTranCount / AlternativeTranCount$$

$$SRR = (AdjustSucceedTranCount + AlterSucceedTranCount) / (AdjustOnceTranCount + AdjustTwiceTranCount + AlternativeTranCount)$$

$$AoAR = (AdjustOnceTranCount + AdjustTwiceTranCount + AlternativeTranCount) / ConflictedOriginalTranCount$$

$$CRR = ConflictReservationCount / ConflictedOriginalTranCount$$

To obtain a more general and accurate estimation of above criteria, we run each test case of Group 6 ten times and collect their average results, as shown in Figure 7.14. Compared with previous simulation results, one more statistical information is collected: *endTime*. *endTime* represents the number of ticks taken from the beginning of a simulation (0 tick) to the end execution time of the last transaction in a schedule, and we consider it as the execution time of a schedule. Besides, the last two columns shows the average values for Case 2-4 and Case 5-7 respectively.

The corresponding values of above evaluation criteria are listed in Table 7.2.

As mentioned in previous section, the result for *ScheduledTranCount*, *ExecutedTranCount*, and *CompleteTranCount* of Case 0 are not useful for our analysis because we're only check the number of conflicting reservations, so above criteria are not applicable to Case 0.

Criteria	Case 0	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	2-4	5-7
ToTT	-	2.5425	2.5183	2.5425	2.5368	2.4457	2.4642	2.4444	2.5325	2.4514
$ToTT_1$	-	2.5425	2.7688	2.7697	2.7931	2.8333	2.94	2.9138	2.7772	2.8958
ToPT	-	2.3004	2.3482	2.3815	2.3613	2.3845	2.3725	2.3612	2.3637	2.3727
ToCT	-	2.3004	2.3482	2.3815	2.3613	2.3845	2.3725	2.3612	2.3637	2.3727
PoS	-	0.9047	0.9324	0.9324	0.9309	0.9750	0.9628	0.9660	0.9333	0.9679
PoE	-	1	1	1	1	1	1	1	1	1
PoTP	-	0.9048	0.9324	0.9324	0.9309	0.9750	0.9628	0.9660	0.9333	0.9679
SOAR	-	-	-	-	-	0.4938	0.4157	0.4570	-	0.4555
STAR	-	-	-	-	-	0.3415	0.2212	0.2673	-	0.2766
SADR	-	-	-	-	-	0.4426	0.3440	0.3902	-	0.3923
SALR	-	-	0.2299	0.2023	0.2211	0.1296	0.1358	0.1351	0.2178	0.1335
SRR	-	-	0.2299	0.2023	0.2211	0.3859	0.2975	0.3380	0.2178	0.3405
AoAR	-	-	1.0809	1.0980	1.0857	1.8395	2.0393	1.9409	1.0882	1.9399
CRR	-	-	1.9538	2.0523	2	2.2222	2.8820	2.5430	2.002	2.5491

Table 7.2: Statistics of Simulation Group 6

Also for Case 1, we’re not creating alternative transactions or adjusted transactions. Fields that are not applicable are skipped in Table 7.2.

Throughput

In Case 1-7, ToTT (Throughput of Triggered Transactions) is 2.53 (transactions per thousand ticks) for Case 2-4, and 2.45 for Case 5-7. Although the number of transactions specified in the scenario files is 188 in all cases, the execution time of the whole schedule for Case 5-7 (76.69 thousand ticks) is a little longer than Case 2-4 (74.23 thousand ticks) because more conflict resolving operations are involved in Case 5-7, and we expect ToTT for Case 5-7 is smaller than Case 2-4. However, if we consider adjusted and alternative transactions as part to triggered transactions ($ToTT_1$), ToTT of Case 5-7 is around 2.89, which is higher than 2.77 for Case 2-4.

ToPT (Throughput of Pre-committed Transactions) and ToCT (Throughput of Committed Transactions) are the same for all cases. This is because we assume that the commit phase is always carried out without exceptions, and *ScheduledTranCount* and *ExecutedTranCount* have the same value in every case. For ToPT, and ToCT, we see that the average

value of Case 5-7 (2.37 transactions per thousand ticks) is slightly higher than Case 2-4 (2.36). Although the average execution time for Case 5-7 is longer than Case 2-4, they have more transactions being scheduled and executed than Case 2-4, which neutralizes the effect caused by longer execution time and makes ToPT and ToCT of Case 5-7 higher than that of Case 2-4.

From above three throughput values, we conclude that Win-Win strategy (Case 5-7) supports better concurrency than Win-Lose strategy (Case 2-4), and it allows more transactions to be scheduled and executed per time unit. However, both strategies perform better than Case 1 when a conflicting transaction is aborted without replacements, and Case 0 when conflict resolving is disabled (refer to Case 0 in Section 7.7.2 for explanation).

Productivity

For PoS (Productivity of Scheduling), since more transactions are scheduled in Case 5-7 than Case 2-4, we expect that PoS for Case 5-7 is higher than Case 2-4, which means higher percentage of triggered transactions will be scheduled in Case 5-7. The result proves our expectation: Case 5-7 has 96.79% of 188 triggered transactions being scheduled, while Case 2-4 has only 93.33%.

PoE (Productivity of Executing) indicates the ratio of executed transactions out of scheduled transactions. Since the commit phase always goes without exceptions, the ratio stays 1 in all cases.

PoTP (Productivity of Transaction Processing) gives ratio of executed transactions to triggered transactions, which gives the same value as PoS because the commit phase is exception-free.

Productivity show the performance of an algorithm in resolving potential conflicts. From above three productivity values, we know that Win-Win strategy performs better than Win-Lose strategy in resolving potential conflicts, which meets our expectation. Still, both strategies out-perform than Case 0 and Case 1.

Algorithm Efficiency

SOAR and STAR show the success rate of the first and second adjustment attempts in resolving transaction conflicts respectively, while SADR (a combination of SOAR and STAR) shows the successful rate of transaction adjustment. These three criteria are only applicable to Case 5-7. From Table 7.2, we see that SOAR (0.45) is higher than STAR (0.27). There are two possible explanations for this. First, the algorithms used for the first adjustment attempt perform better than those for the second attempt. Second, the second adjustment attempt is a substitution for the failing first attempt, which means the second attempt is confronted with a more complicated situation than the first adjustment attempt. Together, in all adjustment attempts, about 39.22% of conflicting transactions are successfully resolved.

SALR shows the success rate of alternative transactions in resolving conflicts, and we see that Case 2-4 has a higher SALR than Case 5-7. From Figure 7.14, we know that the average number of alternative transactions for Case 5-7 (6.96) is much smaller than that of Case 2-4 (18.16). This is because no transaction adjustment is applied in Case 2-4 and alternative transaction is the main strategy to resolve conflicts. In Case 5-7, alternative transaction strategy is applied only when two adjustment attempts fail, which leaves a more complicated situation when an alternative transaction is created in Case 5-7 than Case 2-4, and we expect the harder will it be for an alternative transaction to resolve conflicts in Case 5-7.

SRR (Successful Resolving Rate) considers both adjusted and alternative transactions for the success rate of conflict resolution. In Case 2-4, only alternative transaction is used, and SRR is 21.78%. While in Case 5-7, a 34.05% is achieved. Thus we know, a combination of adjusted and alternative transactions performs better in resolving conflicts than just the alternative transaction strategy.

AoAR (Adjustment or Alternative Rate) shows how many adjusted or alternative transactions are created to resolve a conflicting transaction. In Case 5-7, for every conflicting transaction, an average of 1.94 adjusted or alternative transactions are created to resolve

conflicts. For Case 2-4, the number is 1.08, which is smaller than Case 5-7. This meets our expectation since in Case 5-7, more attempts are carried out to resolve conflicts, which result in more adjusted or alternative transactions being created.

CRR (Conflict Reservation Rate) shows the ratio of the number conflicting reservations detected by the *ResourceServer* to the number of original (non-adjusted and non-alternative) conflicting transactions. One thing to note is that the conflicting reservations are not only from the the original transactions, but also from adjusted or alternative transactions. In Case 5-7, to resolve a conflict, an average of 2.54 conflicting reservations per original conflicting transaction will be caused when attempting to resolve conflicts using adjusted or alternative transactions. In Case 2-4, the number is smaller, 2.00. The same reason as AoAR applies here, i.e., more conflict resolving attempts are made in Case 5-7 than Case 2-4.

A good adjusted or alternative transaction generating method should give higher values for SOAR, STAR, SALR, and SRR, and smaller values for AoAR and CRR. Since in our current implementation, we only implement one *ConflictResolver* component, we don't have a reference to evaluate its method performance. However, as mentioned in Section 7.6.6, we only consider general situations for transaction conflicts when implementing methods of *ConflictResolver*, and we expect that algorithms taking advantage of conflict usages analyzing and machine learning have better performance than ours.

7.7.4 Scalability

To test the scalability of CPSNET, we design four scalability tests based on Group 6. In all these tests, we focus only on Test Case 7, and Test Case 7 of Group 6 (Figure 7.14 and Figure 7.2) is used as the base case, name *Scale 0*.

- In *Scale 0*, we simulate 10 CPS entities with 188 transactions, as shown in Figure 7.11.
- In *Scale 1*, we simulate 10 CPS entities with 390 transactions. These 10 ten entities have the same start system states as those in *Scale 0*, and they repeat the operations

of entities in *Scale 0* two times.

- In *Scale 2*, we simulate 10 CPS entities with 780 transactions. These 10 entities also have the same start system states as those in *Scale 0*, and they repeat twice the operations of entities in *Scale 1*.
- In *Scale 3*, we simulate 20 entities with 376 transactions. The first 10 entities are the same as those in *Scale 0*, and the other 10 entities are created by placing each of the first 10 entities 20 meters ahead. Thus, CPS Entity 11 is 20 meters ahead of CPS Entity 1 in the same lane, CPS Entity 12 is 20 meters ahead of CPS Entity 2 in the same lane, and so on. Transactions of CPS Entities in each pair are the same except the start system states.
- In *Scale 4*, we simulate 40 entities with 752 transactions. Similar strategies are used to create CPS entities based on *Scale 3*.

Each test is run ten times and the average simulation result is collected. Based on the results, the values for evaluation criteria is calculated. The simulation results and values of evaluation criteria are shown in Table 7.3 and Table 7.4 respectively.

Scale in Number of Transactions

Scale 0, *Scale 1* and *Scale 2* have the same number of CPS entities, but the number of transactions is doubled in each test. Since CPS entities in *Scale 1* repeat operations of CPS entities in *Scale 0* twice and *Scale 2* repeats those in *Scale 1* twice, we expect that the increase in the schedule execution time to be also linear, and the simulation result meets our expectation (*EndTime*: 76.91, 154.70, and 313.44 thousand ticks). Although the number of transactions is double, each test only has ten CPS entities and only ten transactions can be executed concurrently at the same time (one for each CPS entity). The simulation result meets our expectation that the number of conflicts will **at most** be doubled in each test (*ConflictedOriginalTranCount* and *ConflictReservationCount*).

Counters	Scale 0	Scale 1	Scale 2	Scale 3	Scale 4
GeneratedTranCount	188	390	780	376	752
ConflictedOriginalTranCount	18.6	24.7	28.4	43.7	115.6
NotAdjustedTranCount	0	0	0	0	0
AdjustOnceTranCount	18.6	24.7	28.3	43.6	115.5
AdjustTwiceTranCount	10.1	14.7	16.9	25.6	71.2
AlternativeTranCount	7.4	9.7	11.7	18.7	50.7
AdjustSucceedTranCount	11.2	15	16.6	24.9	64.8
AdjustFailTranCount	17.5	24.4	28.6	44.3	121.9
AdjustOnceSucceedCount	8.5	10	11.4	18	44.3
AdjustOnceFailCount	10.1	14.7	16.9	25.7	71.2
AdjustTwiceSucceedCount	2.7	5	5.2	6.9	20.5
AdjustTwiceFailCount	7.4	9.7	11.7	18.7	50.7
AlterSucceedTranCount	1	1.2	2.5	3.8	9.2
AlterFailTranCount	6.4	8.5	9.2	14.9	41.5
ScheduledTranCount	181.6	381.5	770.8	361.1	710.5
ExecutedTranCount	181.6	381.5	770.8	361.1	710.5
CompleteTranCount	181.6	381.5	770.8	361.1	710.5
IncompleteTranCount	0	0	0	0	0
PreemptedTranCount	0	0	0	0	0
ConflictReservationCount	47.3	63.8	72	121.1	345.9
NumberOfMessages	672.3	1317.3	2510.7	1391.7	2968.2
EndTime (ticks)	76910	154704.5	313444.1	78144.6	78203.7

Table 7.3: Simulation Result of Scaling Experiments

Criteria	Scale 0	Scale 1	Scale 2	Scale 3	Scale 4
ToTT	2.4444	2.5209	2.4884	4.8116	9.6159
$ToTT_1$	2.9138	2.8383	2.6700	5.9364	12.6516
ToPT	2.3612	2.4660	2.4591	4.6209	9.0852
ToCT	2.3612	2.4660	2.4591	4.6209	9.0852
PoS	0.9660	0.9782	0.9882	0.9604	0.9448
PoE	1	1	1	1	1
PoTP	0.9660	0.9782	0.9882	0.9604	0.9448
SOAR	0.4570	0.4049	0.4028	0.4128	0.3835
STAR	0.2673	0.3401	0.3077	0.2695	0.2879
SADR	0.3902	0.3807	0.3673	0.3598	0.3471
SALR	0.1351	0.1237	0.2137	0.2032	0.1815
SRR	0.3380	0.3299	0.3357	0.3265	0.3117
AoAR	1.9409	1.9879	2.0035	2.0114	2.0536
CRR	2.5430	2.5830	2.5352	2.7711	2.9922

Table 7.4: Statistics of Scaling Experiments

In Table 7.4, we see a stable performance of the transaction processing algorithm in most criteria. For schedule throughput (ToTT, ToPT, and ToCT), although the number of triggered, scheduled and executed transactions doubles in each test, the execution time of the schedule also doubles and it counteracts the influence from the increasing number of transactions. For schedule productivity (PoS, PoE and PoTP) and algorithm efficiency criteria (SOAR, STAR, SADR, SALR, etc.), all three cases have similar values, which indicates that the performance of the scheduling algorithm and conflict resolving algorithms are stable and not affected by the increasing number of transactions.

In all three tests with 30 simulation runs, the CPSNET platform finishes all simulations without errors, which proves that the platform is able to simulate a large load of transactions and operates normally in a long simulation.

Scale in Number of Entities

In *Scale 0*, *Scale 3* and *Scale 4*, both the number of CPS entities and transactions are doubled in each test. The increasing density of cars causes more transaction conflicts, as shown by *ConflictedOriginalTranCount* and *ConflictReservationCount* in Table 7.3. As a result, more adjusted and alternative transactions are created to resolve conflicts. Since the number of transactions executed by each CPS entity is not doubled, we doesn't see an obvious increase in the schedule execution time (*EndTime*: 76.91, 78.14, 78.20 thousand ticks).

In all three tests, the number of CPS simulated entities is doubled in each test, which results in a high level of transaction concurrency, i.e., more transactions are triggered, scheduled and executed concurrently. This is revealed by increasing values for the schedule throughput (ToTT, ToPT, ToCT). For example, the ToTT for *Scale 0* is 2.44 transactions per thousand ticks, the ToTT for *Scale 3* is 4.81 transactions per thousand ticks, and the ToTT for *Scale 4* for 9.62 transactions per thousand ticks. If we consider adjusted and alternative transactions ($ToTT_1$), higher values are observed. Because of the increasing number of transaction conflicts and CPS entity density, the schedule productivity decrease

slightly, e.g., 96.60% of PoS for *Scale 0*, 96.04% of PoS for *Scale 3*, and 94.48% of PoS for *Scale 4*. However, a large drop of the schedule productivity is not seen.

As for algorithm efficiency criteria (SOAR, STAR, SADR, SALR, etc.), we observe relatively stable values in Table 7.4, similar to the case when only the number of transactions is increased. Especially, for each original conflicting transaction, stability is observed in the number of adjusted or alternative transactions (AoAR) that are created to resolve conflicts and the number of conflicting reservations (CRR) that are caused. AoAR and CRR rises slightly in *Scale 3* and *Scale 4* compared with *Scale 0*, even though the number of conflicting transactions (*ConflictedOriginalTranCount*) and reservations (*ConflictReservationCount*) are increased greatly.

The increasing scheduling throughput shows that the simulation platform and the scheduling algorithm scale well in high transaction concurrency environment, and the stable algorithm efficiency values prove that the scheduling and conflict resolving algorithms perform well in an environment of a large load of CPS entities and transactions.

In a total of 20 simulation runs for *Scale 3* and *Scale 4*, the CPSNET platform finishes all simulations without any errors, and it shows that the platform is able to simulate a large load of CPS entities and transactions. Moreover, it performs well as concurrency is increased.

7.8 Summary

In this chapter, we presented how the simulation experiments were carried out using CP-SNET. We described different simulation groups, analyzed the simulation results, and verified different aspects of the simulation platform and the transaction model.

Group 0 shows that our simulation platform carries out a simulation correctly: the *ResourceServer* handles reservation requests as expected, the *Network* delivers messages correctly, and the transaction processing (Figure 6.7) flow is followed by each component of a *CyberPhysicalSystem*.

Groups 1-5 shows how different conflict resolving strategies are applied for different types of transactions. Verification results show that both Win-Lose (Case 2-4) and Win-Win (Case 5-7) strategies are able to resolve a transaction conflict. Compared with no conflict resolving (Case 0) and direct abortion strategy (Case 1), Win-Lose and Win-Win strategies allow more transactions to be scheduled and executed.

Group 6 carries out a comprehensive test on both the simulation platform and the transaction processing algorithms. By running each test case ten times, we presented the average simulation results of Group 6. We evaluate the performance of transaction processing and conflict resolving algorithms using the criteria we have developed. The simulation result proves that our conflicting resolving strategies (Win-Lose and Win-Win) not only allow more transactions to be scheduled and executed successfully, but also maintain a better transaction concurrency level than cases when no conflict resolving is used or direct abortion is applied. As for Win-Lose and Win-Win strategy, Win-Win strategy has a better performance in supporting concurrency and resolving conflicts. In the end, we test the scalability of our simulation platform by increasing the number of CPS entities and transactions to be simulated. Simulation results prove that our platform scales well with the number of CPS entities and transactions, and the scheduling and conflict resolving algorithms give stable performance in an environment of a large load of CPS entities and transactions.

In conclusion, we show that our simulation platform works as expected in carrying out simulations of a CPS Network, which is a distributed system that includes different entities and components communicating through messages, and that the simulation platform scales well and supports simulating a large number of CPS entities and transactions. We also prove that the two-phase commit transaction processing flow is followed by different components of a CPS entity, and by different entities in a CPS Network. As for conflict resolving, our solution (Win-Lose and Win-Win strategies) out-performs solutions where no conflict resolving is applied or direct abortion of a conflicting transaction is used. Our solution not only allows more transactions to be executed, but also maintains a higher transaction

concurrency level.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

Accompanying the promising future of Cyber-Physical Systems in transforming our world with more intelligent and efficient systems, today's technologies are confronted with great challenges to integrate the computational and physical world into a single system. Great endeavors have been made by researchers in different areas to examine current technologies, identify potential challenges, and propose possible solutions for building CPSs that meet different requirements.

In this dissertation, we focus on a particular type of CPSs, Environmental Resource Dependent Cyber-Physical System (ERDCPS). An ERDCPS relies on environmental resources to perform its physical operations, and its operating environment includes other peer ERDCPSs. These ERDCPSs together constitute a CPS Network, within which each ERDCPS competes with another for using environmental resources for their physical operations. The usage of an environmental resource by a physical operation is precise in both time and space. It specifies not only which resource is used, but also which time period the resource is used. Since all ERDCPSs in a CPS Network share the same environmental resources, if operations from more than one ERDCPS access a particular resource at the same time, a conflict occurs.

We propose a transaction model for an ERDCPS. In this model, physical operations of

an ERDCPS are defined in two abstraction levels: action and transaction. A transaction achieves a goal and is composed of a sequence of actions. An action or a transaction requires exclusive access to environmental resources for its operations. Two transactions from different ERDCPSs have a conflict when they use the same resources at the same time. We design a two-phase commit algorithm to process transactions in an ERDCPS. In the pre-commit phase, a transaction is scheduled and potential transaction conflicts are detected and resolved. In the commit phase, a pre-committed transaction is executed. We propose several conflict resolving and exception handling strategies to avoid transaction conflicts in both the pre-commit and commit phases. Two general algorithms that work in the centralized and distributed resource management environments respectively are presented.

To simulate the transaction model, we develop a highly configurable and modularized simulation platform: CPSNET. CPSNET supports configuration of the simulation environment, CPS entities and the two-phase commit algorithm. Various statistical information and operation logs are provided to monitor and evaluate the platform itself and the transaction model. CPSNET performs a time-based simulation and it simulates a CPS Network consisting of CPS entities, an optional Resource Server, and a Communication Network. Each of these entities operates independently and communicates with others using messages. In each CPS entity, the two-phase commit algorithm is used to process transactions.

Several groups of simulations are run using the CPSNET platform to verify the platform itself and the transaction model. Simulation results show that the simulation platform always carries out a simulation successfully, and each entity does its job as expected. For a CPS entity, all components fulfill their functions correctly with respect to the two-phase commit transaction processing algorithm. Simulation results also show that the proposed conflict resolving strategies improve schedule throughput and productivity compared with cases where no conflict resolving strategy is used or a conflicting transaction is directly aborted. Moreover, the platform is proved to scale well and support simulating a large load of CPS entities and transactions.

This dissertation contributes to the research of CPSs in several main aspects. First, it explicitly defines ERDCPS, a particular application area of CPSs, and its operating environment, CPS Network. Second, it proposes a new transaction model for CPS with both definitions and algorithms. Third, a simulation platform, CPSNET, is developed to simulate the transaction model.

8.2 Future Work

The research work presented in this dissertation has raised possibilities of future work in many areas.

First, we plan to add more application examples, e.g., traffic management systems [53–55], to demonstrate the model, and show how the transaction model can be applied to different application areas of CPSs.

Second, the two-phase transaction processing algorithm can be enhanced in several aspects.

1. More complex conflict resolving and exception handling strategies can be designed to improve the schedule throughput and productivity. We know from simulations presented in Chapter 7 that reservation requests are served in FCFS (First come, first served) manner, and randomness of the scheduling process causes different simulation results in different runs of the same test case. Assuming that a transaction T_1 from CPS entity E_1 has conflicts with a transaction T_2 from CPS entity E_2 . Aborting (or adjusting) T_1 results in a different schedule from aborting (or adjusting) T_2 . We can improve our algorithm by aborting or adjusting the transaction that creates a schedule with higher throughput and productivity. To achieve this, batch processing of reservation requests is needed. In a fixed time range, n pending reservations are checked, and conflicting transactions that cause less chaining conflicts are selected to be adjusted.
2. Transaction adjustment methods defined in the Conflict Resolver component consider

only general conflict scenarios, which causes a relative low rate of successful adjustments (Table 7.2). An improvement will be taking conflicting resource usages and transaction types into account, and utilizing techniques such as machine learning [59] to select an adjustment that has a higher probability to resolve potential conflicts.

3. In centralized resource management environment, the Resource Server can extend its functionality from detecting potential conflicts to others such as ensuring resource usage fairness among CPS entities, achieving a higher resource usage rate, and increasing schedule throughput, which are implemented by resource scheduling algorithms in traditional real-time systems [48].

Third, in simulations we have performed, the commit phase is neglected. In the future development, the simulation of the commit phase will be added, and an exception generator will be developed to create random exceptions in order to test the performance of the exception handling algorithms. Moreover, emergency handling transactions will also be generated to test the transaction preemption strategy.

Fourth, the simulations we have performed are in centralized mode. Simulation in distributed mode will be performed in the future to test the two-phase commit algorithm in distributed resource management environment.

Last, several extensions can be added to the the CPSNET simulation platform.

1. For each type of component, add more instances that use different algorithms to fulfill the required function. For example, we can enhance the scheduling algorithm of the pre-commit phase by developing new instances of the Transaction Scheduler component. Moreover, dependency injection techniques [60, 61] and service-oriented architecture [29, 30] can be introduced to enable dynamic selection of proper component instance for a simulation.
2. Extend statistical methods to allow verification of customized properties. For example, if we need to verify that a schedule of transaction does not have any conflicting

transactions, we can develop assertion methods that check the conflicting transaction count, and tell whether the specified property is maintained.

3. Develop a front-end GUI [62] for configuring the simulation platform and CPS entities, and for showing statistical information collected in the simulation result.
4. Open source the simulation platform, and make it available to other researchers.

Bibliography

- [1] NSF CPS Program, <http://www.nsf.gov/pubs/2011/nsf11516/nsf11516.htm>
- [2] NSF CPS Workshop, <http://varma.ece.cmu.edu/cps/>
- [3] Berkeley CHESS CPS Group, <http://cyberphysicalsystems.org/>
- [4] Lee, Edward A. “Computing foundations and practice for cyber-physical systems: A preliminary report.” *University of California, Berkeley, Tech. Rep. UCB/EECS-2007-72* (2007).
- [5] Lee, Edward A. “CPS foundations.” In *Proceedings of the 47th Design Automation Conference*, pp. 737-742. ACM, 2010.
- [6] Sha, Lui, Sathish Gopalakrishnan, Xue Liu, and Qixin Wang. “Cyber-physical systems: A new frontier.” In *Machine Learning in Cyber Trust*, pp. 3-13. Springer US, 2009.
- [7] Rajkumar, Ragnathan Raj, Insup Lee, Lui Sha, and John Stankovic. “Cyber-physical systems: the next computing revolution.” In *Proceedings of the 47th Design Automation Conference*, pp. 731-736. ACM, 2010.
- [8] Baheti, Radhakisan, and Helen Gill. “Cyber-physical systems.” *The impact of control technology* (2011): 161-166.
- [9] *Proceedings of The IEEE*, Special Issue on Cyber-Physical Systems, vol.100, no.1, Jan 2012
- [10] Kim, Kyoung-Dae, and Panganamala R. Kumar. “Cyberphysical systems: A perspective at the centennial.” *Proceedings of the IEEE* 100, no. Special Centennial Issue (2012): 1287-1308.

- [11] Wang, Yunbo, Mehmet C. Vuran, and Steve Goddard. “Cyber-physical systems in industrial process control.” *ACM SIGBED Review* 5, no. 1 (2008): 12.
- [12] Lee, Insup, Oleg Sokolsky, Sanjian Chen, John Hatcliff, Eunkyong Jee, BaekGyu Kim, Andrew King et al. “Challenges and research directions in medical cyberphysical systems.” *Proceedings of the IEEE* 100, no. 1 (2012): 75-90.
- [13] Cheng, Albert MK. “Cyber-physical medical and medication systems.” In *Distributed Computing Systems Workshops, 2008. ICDCS'08. 28th International Conference on*, pp. 529-532. IEEE, 2008.
- [14] Lee, Insup, and Oleg Sokolsky. “Medical cyber physical systems.” In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pp. 743-748. IEEE, 2010.
- [15] Gokhale, Aniruddha, Mark P. McDonald, Steven Drager, and William McKeever. “A cyber physical systems perspective on the real-time and reliable dissemination of information in intelligent transportation systems.” AIR FORCE RESEARCH LAB ROME NY, 2010.
- [16] Abid, Hassan, Luong Thi Thu Phuong, Jin Wang, Sungyoung Lee, and Saad Qaisar. “V-Cloud: vehicular cyber-physical systems and cloud computing.” In *Proceedings of the 4th International Symposium on Applied Sciences in Biomedical and Communication Technologies*, p. 165. ACM, 2011.
- [17] Zhao, Sha, Shijian Li, Longbiao Chen, Yang Ding, Zeming Zheng, and Gang Pan. “iCPS-Car: An Intelligent Cyber-physical System for Smart Automobiles.” In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCoM), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, pp. 818-825. IEEE, 2013.
- [18] Yongfu, Li, Sun Dihua, Liu Weining, and Zhang Xuebo. “A service-oriented architecture for the transportation Cyber-Physical Systems.” In *Control Conference (CCC)*,

2012 31st Chinese, pp. 7674-7678. IEEE, 2012.

- [19] Susuki, Yoshihiko, T. John Koo, Hiroaki Ebina, Takuya Yamazaki, Takashi Ochi, Takuji Uemura, and Takashi Hikihara. "A hybrid system approach to the analysis and design of power grid dynamic performance." *Proceedings of the IEEE* 100, no. 1 (2012): 225-239.
- [20] Tham, Chen-Khong, and Tie Luo. "Sensing-driven energy purchasing in smart grid cyber-physical system." *Systems, Man, and Cybernetics: Systems, IEEE Transactions on* 43, no. 4 (2013): 773-784.
- [21] Lin, Jing, Sahra Sedigh, and Ann Miller. "Towards integrated simulation of cyber-physical systems: a case study on intelligent water distribution." In *Dependable, Autonomous and Secure Computing, 2009. DASC'09. Eighth IEEE International Conference on*, pp. 690-695. IEEE, 2009.
- [22] Zhang, Wei, Maryam Kamgarpour, Dengfeng Sun, and Claire J. Tomlin. "A hierarchical flight planning framework for air traffic management." *Proceedings of the IEEE* 100, no. 1 (2012): 179-194.
- [23] Fink, Jonathan, Alejandro Ribeiro, and Vijay Kumar. "Robust control for mobility and wireless communication in cyberphysical systems with application to robot teams." *Proceedings of the IEEE* 100, no. 1 (2012): 164-178.
- [24] Talcott, Carolyn. "Cyber-physical systems and events." In *Software-Intensive Systems and New Computing Paradigms*, pp. 101-115. Springer Berlin Heidelberg, 2008.
- [25] Tan, Ying, Mehmet C. Vuran, and Steve Goddard. "Spatio-temporal event model for cyber-physical systems." In *Distributed Computing Systems Workshops, 2009. ICDCS Workshops' 09. 29th IEEE International Conference on*, pp. 44-50. IEEE, 2009.

- [26] Tan, Ying, Mehmet C. Vuran, Steve Goddard, Yue Yu, Miao Song, and Shangping Ren. “A concept lattice-based event model for Cyber-Physical Systems.” In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-physical Systems*, pp. 50-60. ACM, 2010.
- [27] Yue, Ke, Li Wang, Shangping Ren, Xufei Mao, and Xiangyang Li. “An adaptive discrete event model for cyber-physical system.” In *Analytic Virtual Integration of Cyber-Physical Systems Workshop, USA*, pp. 9-15. 2010.
- [28] Zhang, Yuanfang, Christopher Gill, and Chenyang Lu. “Reconfigurable real-time middleware for distributed cyber-physical systems with aperiodic events.” In *Distributed Computing Systems, 2008. ICDCS’08. The 28th International Conference on*, pp. 581-588. IEEE, 2008.
- [29] Huang, Jian, Farokh Bastani, I-Ling Yen, and Jun-Jang Jeng. “Toward a smart cyber-physical space: a context-sensitive resource-explicit service model.” In *Computer Software and Applications Conference, 2009. COMPSAC’09. 33rd Annual IEEE International*, vol. 2, pp. 122-127. IEEE, 2009.
- [30] Huang, Jian, F. Bastani, I-L. Yen, Jing Dong, Wenke Zhang, F-J. Wang, and H-J. Hsu. “Extending service model to build an effective service composition framework for cyber-physical systems.” In *Service-Oriented Computing and Applications (SOCA), 2009 IEEE International Conference on*, pp. 1-8. IEEE, 2009.
- [31] Huang, Jian, Farokh B. Bastani, I-Ling Yen, and Wenke Zhang. “A framework for efficient service composition in cyber-physical systems.” In *Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on*, pp. 291-298. IEEE, 2010.
- [32] La, Hyun Jung, and Soo Dong Kim. “A service-based approach to designing cyber

- physical systems.” In *Computer and Information Science (ICIS), 2010 IEEE/ACIS 9th International Conference on*, pp. 895-900. IEEE, 2010.
- [33] Talcott, Carolyn L. “A formal framework for interactive agents.” *Electronic Notes in Theoretical Computer Science* 203, no. 3 (2008): 95-106.
- [34] Lin, Jing, Sahra Sedigh, and Ann Miller. “Modeling cyber-physical systems with semantic agent.” In *Computer Software and Applications Conference Workshops (COMP-SACW), 2010 IEEE 34th Annual*, pp. 13-18. IEEE, 2010.
- [35] Zhu, Quanyan, Linda Bushnell, and Tamer Baar. “Resilient Distributed Control of Multi-agent Cyber-Physical Systems.” In *Control of Cyber-Physical Systems*, pp. 301-316. Springer International Publishing, 2013.
- [36] Derler, Patricia, Edward A. Lee, and A. Sangiovanni Vincentelli. “Modeling cyberphysical systems.” *Proceedings of the IEEE* 100, no. 1 (2012): 13-28.
- [37] Sztipanovits, Janos, Xenofon Koutsoukos, Gabor Karsai, Nicholas Kottenstette, Panos Antsaklis, Vijay Gupta, Bill Goodwine, John Baras, and Shige Wang. “Toward a science of cyberphysical system integration.” *Proceedings of the IEEE* 100, no. 1 (2012): 29-44.
- [38] Madria, Sanjay Kumar, and Bharat Bhargava. “A transaction model to improve data availability in mobile computing.” *Distributed and Parallel Databases* 10, no. 2 (2001): 127-160.
- [39] Mehrotra, Sharad, Rajeev Rastogi, Abraham Silberschatz, and Henry F. Korth. “A transaction model for multidatabase systems.” In *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*, pp. 56-63. IEEE, 1992.
- [40] Serrano-Alvarado, Patricia, Claudia Roncancio, and Michel Adiba. “A survey of mobile transactions.” *Distributed and Parallel databases* 16, no. 2 (2004): 193-230.

- [41] Bernstein, Philip A., and Nathan Goodman. "Concurrency control in distributed database systems." *ACM Computing Surveys (CSUR)* 13, no. 2 (1981): 185-221.
- [42] Bhargava, Bharat. "Concurrency control in database systems." *Knowledge and Data Engineering, IEEE Transactions on* 11, no. 1 (1999): 3-16.
- [43] Barghouti, Naser S., and Gail E. Kaiser. "Concurrency control in advanced database applications." *ACM Computing Surveys (CSUR)* 23, no. 3 (1991): 269-317.
- [44] Thomasian, Alexander. "Concurrency control: methods, performance, and analysis." *ACM Computing Surveys (CSUR)* 30, no. 1 (1998): 70-119.
- [45] Garcia-Molina, Hector. "Using semantic knowledge for transaction processing in a distributed database." *ACM Transactions on Database Systems (TODS)* 8, no. 2 (1983): 186-213.
- [46] Farrag, Abdel Aziz, and M. Tamer Ozsu. "Using semantic knowledge of transactions to increase concurrency." *ACM Transactions on Database Systems (TODS)* 14, no. 4 (1989): 503-525.
- [47] Mehrotra, Sharad, Rajeev Rastogi, Yuri Breitbart, Henry F. Korth, and Avi Silber-schatz. "The concurrency control problem in multidatabases: Characteristics and solutions." In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data (SIGMOD '92)*, pp.288-297, 1992. 2. ACM, 1992.
- [48] Sha, Lui, Tarek Abdelzaher, Karl-Erik Arzen, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. "Real time scheduling theory: A historical perspective." *Real-time systems* 28, no. 2-3 (2004): 101-155.

- [49] Parikh, Swapnil M. “A survey on cloud computing resource allocation techniques.” In *Engineering (NUiCONE), 2013 Nirma University International Conference on*, pp. 1-5. IEEE, 2013.
- [50] Sharma, Raksha, Vishnu Kant Soni, Manoj Kumar Mishra, and Prachet Bhuyan. “A survey of job scheduling and resource management in grid computing.” *world academy of science, engineering and technology* 64 (2010): 461-466.
- [51] Li, Qiao. “Scheduling in cyber-physical systems.” PhD Thesis (2012).
- [52] Ghodsi, Ali, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. “Dominant Resource Fairness: Fair Allocation of Multiple Resource Types.” In *NSDI*, vol. 11, pp. 24-24. 2011.
- [53] Azimi, Reza, Gaurav Bhatia, Raj Rajkumar, and Priyantha Mudalige. “Intersection management using vehicular networks.” No. 2012-01-0292. SAE Technical Paper, 2012.
- [54] Zhang, Wei, Maryam Kamgarpour, Dengfeng Sun, and Claire J. Tomlin. “A hierarchical flight planning framework for air traffic management.” *Proceedings of the IEEE* 100, no. 1 (2012): 179-194.
- [55] Merah, Amar Farouk, Samer Samarah, and Azzedine Boukerche. “Vehicular movement patterns: a prediction-based route discovery technique for VANETs.” In *Communications (ICC), 2012 IEEE International Conference on*, pp. 5291-5295. IEEE, 2012.
- [56] Dedicated Short-Range Communications (DSRC), <http://www.its.dot.gov/DSRC/>
- [57] Zhu, Huang, and Gurdip Singh. “A communication protocol for a vehicle collision warning system.” In *Proceedings of the 2010 IEEE/ACM Int’l Conference on Green Computing and Communications & Int’l Conference on Cyber, Physical and Social Computing*, pp. 636-644. IEEE Computer Society, 2010.
- [58] The Java Programming Language, [http://en.wikipedia.org/wiki/Java_\(programming_language\)](http://en.wikipedia.org/wiki/Java_(programming_language))

[59] Machine Learning, http://en.wikipedia.org/wiki/Machine_learning

[60] Inversion of Control, http://en.wikipedia.org/wiki/Inversion_of_control

[61] Spring Framework, <http://projects.spring.io/spring-framework/>

[62] Java Swing, [http://en.wikipedia.org/wiki/Swing_\(Java\)](http://en.wikipedia.org/wiki/Swing_(Java))

Appendix A

Sample Simulation Result of CyberPhysicalSystem

```
##### Simulation Report of CPS_ENTITY_1
  Group: GROUP_6, Case: CASE_8
  Date of Simulation: Sun Jun 01 02:19:00 CDT 2014
  ##### GENREAL INFORMATION
  CPS Entity ID: CPS_ENTITY_1
  Start State: {Speed: 2.0, Acceleration: 0.0, Location: <ROAD_1,1,0>}
  End State: {Speed: 4.0, Acceleration: 0.0, Location: <ROAD_1,1,376>}
  Execution Time: [0, 75152)
  ##### TRANSACTIONS
  # Number of transactions in finishQueue is: 26
  ## Transactions in finishQueue: 26
  Transaction: ConstantSpeedTransaction: [0, 2000), {ConstantSpeedAction: [0, 2000)}
  Resource Usage:
  <ROAD_1,1,0>: [0, 500)
  <ROAD_1,1,1>: [500, 1000)
  <ROAD_1,1,2>: [1000, 1500)
  <ROAD_1,1,3>: [1500, 2000)
  Transaction: AccelerateTransaction: [2000, 5000), {IncreaseSpeedAction: [2000, 5000), {targetSpeed: 4.0}}
  Resource Usage:
  <ROAD_1,1,4>: [2000, 2465)
  <ROAD_1,1,5>: [2465, 2874)
  <ROAD_1,1,6>: [2874, 3244)
  <ROAD_1,1,7>: [3244, 3584)
  <ROAD_1,1,8>: [3584, 3901)
  <ROAD_1,1,9>: [3901, 4199)
  <ROAD_1,1,10>: [4199, 4481)
  <ROAD_1,1,11>: [4481, 4749)
  <ROAD_1,1,12>: [4749, 5000)
  Transaction: ChangeToRightLaneTransaction: [5000, 9001), {ConstantSpeedAction: [5000, 6000), TurnRightAction:
[6000, 8001), {angle: 30}, ConstantSpeedAction: [8001, 9001)}
  Resource Usage:
  <ROAD_1,1,13>: [5000, 5250)
  <ROAD_1,1,14>: [5250, 5500)
  <ROAD_1,1,15>: [5500, 5750)
  <ROAD_1,1,16>: [5750, 6000)
  <ROAD_1,1,17>: [6000, 6289)
  <ROAD_1,1,18>: [6289, 6578)
  <ROAD_1,1,19>: [6578, 6667)
  <ROAD_1,1,20>: [6667, 6956)
  <ROAD_1,2,20>: [6667, 6956)
  <ROAD_1,1,21>: [6956, 7245)
  <ROAD_1,2,21>: [6956, 7245)
```

<ROAD_1,1,22>: [7245, 7334]
 <ROAD_1,2,22>: [7245, 7334]
 <ROAD_1,2,23>: [7334, 7623]
 <ROAD_1,2,24>: [7623, 7912]
 <ROAD_1,2,25>: [7912, 8001]
 <ROAD_1,2,26>: [8001, 8251]
 <ROAD_1,2,27>: [8251, 8501]
 <ROAD_1,2,28>: [8501, 8751]
 <ROAD_1,2,29>: [8751, 9001]
 Transaction: AccelerateTransaction: [9001, 12001], {IncreaseSpeedAction: [9001, 12001], {targetSpeed: 6.0}}
 Resource Usage:
 <ROAD_1,2,30>: [9001, 9246]
 <ROAD_1,2,31>: [9246, 9482]
 <ROAD_1,2,32>: [9482, 9710]
 <ROAD_1,2,33>: [9710, 9930]
 <ROAD_1,2,34>: [9930, 10144]
 <ROAD_1,2,35>: [10144, 10352]
 <ROAD_1,2,36>: [10352, 10554]
 <ROAD_1,2,37>: [10554, 10751]
 <ROAD_1,2,38>: [10751, 10943]
 <ROAD_1,2,39>: [10943, 11130]
 <ROAD_1,2,40>: [11130, 11313]
 <ROAD_1,2,41>: [11313, 11492]
 <ROAD_1,2,42>: [11492, 11667]
 <ROAD_1,2,43>: [11667, 11839]
 <ROAD_1,2,44>: [11839, 12001]
 Transaction: ConstantSpeedTransaction: [12001, 14001], {ConstantSpeedAction: [12001, 14001]}
 Resource Usage:
 <ROAD_1,2,45>: [12001, 12168]
 <ROAD_1,2,46>: [12168, 12335]
 <ROAD_1,2,47>: [12335, 12502]
 <ROAD_1,2,48>: [12502, 12669]
 <ROAD_1,2,49>: [12669, 12836]
 <ROAD_1,2,50>: [12836, 13003]
 <ROAD_1,2,51>: [13003, 13170]
 <ROAD_1,2,52>: [13170, 13337]
 <ROAD_1,2,53>: [13337, 13504]
 <ROAD_1,2,54>: [13504, 13671]
 <ROAD_1,2,55>: [13671, 13838]
 <ROAD_1,2,56>: [13838, 14001]
 Transaction: ChangeToRightLaneTransaction: [14001, 18144], {IncreaseSpeedAction: [14001, 15001], {targetSpeed: 7.0},
 ConstantSpeedAction: [15001, 16001], TurnRightAction: [16001, 17144], {angle: 30}, ConstantSpeedAction: [17144, 18144]}
 Resource Usage:
 <ROAD_1,2,57>: [14001, 14166]
 <ROAD_1,2,58>: [14166, 14327]
 <ROAD_1,2,59>: [14327, 14484]
 <ROAD_1,2,60>: [14484, 14637]
 <ROAD_1,2,61>: [14637, 14787]
 <ROAD_1,2,62>: [14787, 14933]
 <ROAD_1,2,63>: [14933, 15001]
 <ROAD_1,2,64>: [15001, 15144]
 <ROAD_1,2,65>: [15144, 15287]
 <ROAD_1,2,66>: [15287, 15430]
 <ROAD_1,2,67>: [15430, 15573]
 <ROAD_1,2,68>: [15573, 15716]
 <ROAD_1,2,69>: [15716, 15859]
 <ROAD_1,2,70>: [15859, 16001]
 <ROAD_1,2,71>: [16001, 16166]
 <ROAD_1,2,72>: [16166, 16331]
 <ROAD_1,2,73>: [16331, 16382]
 <ROAD_1,2,74>: [16382, 16547]
 <ROAD_1,3,74>: [16382, 16547]
 <ROAD_1,2,75>: [16547, 16712]
 <ROAD_1,3,75>: [16547, 16712]
 <ROAD_1,2,76>: [16712, 16763]

<ROAD_1,3,76>: [16712, 16763]
 <ROAD_1,3,77>: [16763, 16928]
 <ROAD_1,3,78>: [16928, 17093]
 <ROAD_1,3,79>: [17093, 17144]
 <ROAD_1,3,80>: [17144, 17287]
 <ROAD_1,3,81>: [17287, 17430]
 <ROAD_1,3,82>: [17430, 17573]
 <ROAD_1,3,83>: [17573, 17716]
 <ROAD_1,3,84>: [17716, 17859]
 <ROAD_1,3,85>: [17859, 18002]
 <ROAD_1,3,86>: [18002, 18144]
 Transaction: DecelerateTransaction: [18144, 21144), {DecreaseSpeedAction: [18144, 21144), {targetSpeed: 4.0}}
 Resource Usage:
 <ROAD_1,3,87>: [18144, 18289]
 <ROAD_1,3,88>: [18289, 18437]
 <ROAD_1,3,89>: [18437, 18588]
 <ROAD_1,3,90>: [18588, 18743]
 <ROAD_1,3,91>: [18743, 18902]
 <ROAD_1,3,92>: [18902, 19065]
 <ROAD_1,3,93>: [19065, 19232]
 <ROAD_1,3,94>: [19232, 19404]
 <ROAD_1,3,95>: [19404, 19581]
 <ROAD_1,3,96>: [19581, 19764]
 <ROAD_1,3,97>: [19764, 19954]
 <ROAD_1,3,98>: [19954, 20151]
 <ROAD_1,3,99>: [20151, 20356]
 <ROAD_1,3,100>: [20356, 20570]
 <ROAD_1,3,101>: [20570, 20794]
 <ROAD_1,3,102>: [20794, 21030]
 <ROAD_1,3,103>: [21030, 21144]
 Transaction: ConstantSpeedTransaction: [21144, 23144), {ConstantSpeedAction: [21144, 23144)}
 Resource Usage:
 <ROAD_1,3,104>: [21144, 21394]
 <ROAD_1,3,105>: [21394, 21644]
 <ROAD_1,3,106>: [21644, 21894]
 <ROAD_1,3,107>: [21894, 22144]
 <ROAD_1,3,108>: [22144, 22394]
 <ROAD_1,3,109>: [22394, 22644]
 <ROAD_1,3,110>: [22644, 22894]
 <ROAD_1,3,111>: [22894, 23144]
 Transaction: ChangeToRightLaneTransaction: [23144, 27145), {ConstantSpeedAction: [23144, 24144), TurnRightAction:
 [24144, 26145), {angle: 30}, ConstantSpeedAction: [26145, 27145)}
 Resource Usage:
 <ROAD_1,3,112>: [23144, 23394]
 <ROAD_1,3,113>: [23394, 23644]
 <ROAD_1,3,114>: [23644, 23894]
 <ROAD_1,3,115>: [23894, 24144]
 <ROAD_1,3,116>: [24144, 24433]
 <ROAD_1,3,117>: [24433, 24722]
 <ROAD_1,3,118>: [24722, 24811]
 <ROAD_1,3,119>: [24811, 25100]
 <ROAD_1,4,119>: [24811, 25100]
 <ROAD_1,3,120>: [25100, 25389]
 <ROAD_1,4,120>: [25100, 25389]
 <ROAD_1,3,121>: [25389, 25478]
 <ROAD_1,4,121>: [25389, 25478]
 <ROAD_1,4,122>: [25478, 25767]
 <ROAD_1,4,123>: [25767, 26056]
 <ROAD_1,4,124>: [26056, 26145]
 <ROAD_1,4,125>: [26145, 26395]
 <ROAD_1,4,126>: [26395, 26645]
 <ROAD_1,4,127>: [26645, 26895]
 <ROAD_1,4,128>: [26895, 27145]
 Transaction: AccelerateTransaction: [27145, 30145), {IncreaseSpeedAction: [27145, 30145), {targetSpeed: 6.0}}
 Resource Usage:

<ROAD_1,4,129>: [27145, 27390)
 <ROAD_1,4,130>: [27390, 27626)
 <ROAD_1,4,131>: [27626, 27854)
 <ROAD_1,4,132>: [27854, 28074)
 <ROAD_1,4,133>: [28074, 28288)
 <ROAD_1,4,134>: [28288, 28496)
 <ROAD_1,4,135>: [28496, 28698)
 <ROAD_1,4,136>: [28698, 28895)
 <ROAD_1,4,137>: [28895, 29087)
 <ROAD_1,4,138>: [29087, 29274)
 <ROAD_1,4,139>: [29274, 29457)
 <ROAD_1,4,140>: [29457, 29636)
 <ROAD_1,4,141>: [29636, 29811)
 <ROAD_1,4,142>: [29811, 29983)
 <ROAD_1,4,143>: [29983, 30145)
 Transaction: ConstantSpeedTransaction: [30145, 32145), {ConstantSpeedAction: [30145, 32145)}
 Resource Usage:
 <ROAD_1,4,144>: [30145, 30312)
 <ROAD_1,4,145>: [30312, 30479)
 <ROAD_1,4,146>: [30479, 30646)
 <ROAD_1,4,147>: [30646, 30813)
 <ROAD_1,4,148>: [30813, 30980)
 <ROAD_1,4,149>: [30980, 31147)
 <ROAD_1,4,150>: [31147, 31314)
 <ROAD_1,4,151>: [31314, 31481)
 <ROAD_1,4,152>: [31481, 31648)
 <ROAD_1,4,153>: [31648, 31815)
 <ROAD_1,4,154>: [31815, 31982)
 <ROAD_1,4,155>: [31982, 32145)
 Transaction: ChangeToRightLaneTransaction: [32145, 35480), {ConstantSpeedAction: [32145, 33145), TurnRightAction: [33145, 34480), {angle: 30}, ConstantSpeedAction: [34480, 35480)}
 Resource Usage:
 <ROAD_1,4,156>: [32145, 32312)
 <ROAD_1,4,157>: [32312, 32479)
 <ROAD_1,4,158>: [32479, 32646)
 <ROAD_1,4,159>: [32646, 32813)
 <ROAD_1,4,160>: [32813, 32980)
 <ROAD_1,4,161>: [32980, 33145)
 <ROAD_1,4,162>: [33145, 33338)
 <ROAD_1,4,163>: [33338, 33531)
 <ROAD_1,4,164>: [33531, 33590)
 <ROAD_1,4,165>: [33590, 33783)
 <ROAD_1,5,165>: [33590, 33783)
 <ROAD_1,4,166>: [33783, 33976)
 <ROAD_1,5,166>: [33783, 33976)
 <ROAD_1,4,167>: [33976, 34035)
 <ROAD_1,5,167>: [33976, 34035)
 <ROAD_1,5,168>: [34035, 34228)
 <ROAD_1,5,169>: [34228, 34421)
 <ROAD_1,5,170>: [34421, 34480)
 <ROAD_1,5,171>: [34480, 34647)
 <ROAD_1,5,172>: [34647, 34814)
 <ROAD_1,5,173>: [34814, 34981)
 <ROAD_1,5,174>: [34981, 35148)
 <ROAD_1,5,175>: [35148, 35315)
 <ROAD_1,5,176>: [35315, 35480)
 Transaction: DecelerateTransaction: [35480, 38480), {DecreaseSpeedAction: [35480, 38480), {targetSpeed: 4.0}}
 Resource Usage:
 <ROAD_1,5,177>: [35480, 35649)
 <ROAD_1,5,178>: [35649, 35821)
 <ROAD_1,5,179>: [35821, 35996)
 <ROAD_1,5,180>: [35996, 36175)
 <ROAD_1,5,181>: [36175, 36358)
 <ROAD_1,5,182>: [36358, 36545)
 <ROAD_1,5,183>: [36545, 36737)

<ROAD_1,5,184>: [36737, 36934)
 <ROAD_1,5,185>: [36934, 37136)
 <ROAD_1,5,186>: [37136, 37344)
 <ROAD_1,5,187>: [37344, 37558)
 <ROAD_1,5,188>: [37558, 37778)
 <ROAD_1,5,189>: [37778, 38006)
 <ROAD_1,5,190>: [38006, 38242)
 <ROAD_1,5,191>: [38242, 38480)
 Transaction: ConstantSpeedTransaction: [38480, 40480), {ConstantSpeedAction: [38480, 40480)}
 Resource Usage:
 <ROAD_1,5,192>: [38480, 38730)
 <ROAD_1,5,193>: [38730, 38980)
 <ROAD_1,5,194>: [38980, 39230)
 <ROAD_1,5,195>: [39230, 39480)
 <ROAD_1,5,196>: [39480, 39730)
 <ROAD_1,5,197>: [39730, 39980)
 <ROAD_1,5,198>: [39980, 40230)
 <ROAD_1,5,199>: [40230, 40480)
 Transaction: ChangeToLeftLaneTransaction: [40480, 44481), {ConstantSpeedAction: [40480, 41480), TurnLeftAction: [41480, 43481), {angle: 30}, ConstantSpeedAction: [43481, 44481)}
 Resource Usage:
 <ROAD_1,5,200>: [40480, 40730)
 <ROAD_1,5,201>: [40730, 40980)
 <ROAD_1,5,202>: [40980, 41230)
 <ROAD_1,5,203>: [41230, 41480)
 <ROAD_1,5,204>: [41480, 41769)
 <ROAD_1,5,205>: [41769, 42058)
 <ROAD_1,5,206>: [42058, 42147)
 <ROAD_1,5,207>: [42147, 42436)
 <ROAD_1,4,207>: [42147, 42436)
 <ROAD_1,5,208>: [42436, 42725)
 <ROAD_1,4,208>: [42436, 42725)
 <ROAD_1,5,209>: [42725, 42814)
 <ROAD_1,4,209>: [42725, 42814)
 <ROAD_1,4,210>: [42814, 43103)
 <ROAD_1,4,211>: [43103, 43392)
 <ROAD_1,4,212>: [43392, 43481)
 <ROAD_1,4,213>: [43481, 43731)
 <ROAD_1,4,214>: [43731, 43981)
 <ROAD_1,4,215>: [43981, 44231)
 <ROAD_1,4,216>: [44231, 44481)
 Transaction: AccelerateTransaction: [44481, 47481), {IncreaseSpeedAction: [44481, 47481), {targetSpeed: 6.0}}
 Resource Usage:
 <ROAD_1,4,217>: [44481, 44726)
 <ROAD_1,4,218>: [44726, 44962)
 <ROAD_1,4,219>: [44962, 45190)
 <ROAD_1,4,220>: [45190, 45410)
 <ROAD_1,4,221>: [45410, 45624)
 <ROAD_1,4,222>: [45624, 45832)
 <ROAD_1,4,223>: [45832, 46034)
 <ROAD_1,4,224>: [46034, 46231)
 <ROAD_1,4,225>: [46231, 46423)
 <ROAD_1,4,226>: [46423, 46610)
 <ROAD_1,4,227>: [46610, 46793)
 <ROAD_1,4,228>: [46793, 46972)
 <ROAD_1,4,229>: [46972, 47147)
 <ROAD_1,4,230>: [47147, 47319)
 <ROAD_1,4,231>: [47319, 47481)
 Transaction: ConstantSpeedTransaction: [47481, 49481), {ConstantSpeedAction: [47481, 49481)}
 Resource Usage:
 <ROAD_1,4,232>: [47481, 47648)
 <ROAD_1,4,233>: [47648, 47815)
 <ROAD_1,4,234>: [47815, 47982)
 <ROAD_1,4,235>: [47982, 48149)
 <ROAD_1,4,236>: [48149, 48316)

<ROAD_1,4,237>: [48316, 48483)
 <ROAD_1,4,238>: [48483, 48650)
 <ROAD_1,4,239>: [48650, 48817)
 <ROAD_1,4,240>: [48817, 48984)
 <ROAD_1,4,241>: [48984, 49151)
 <ROAD_1,4,242>: [49151, 49318)
 <ROAD_1,4,243>: [49318, 49481)
 Transaction: ChangeToLeftLaneTransaction: [49481, 52816), {ConstantSpeedAction: [49481, 50481), TurnLeftAction: [50481, 51816), {angle: 30}, ConstantSpeedAction: [51816, 52816)}
 Resource Usage:
 <ROAD_1,4,244>: [49481, 49648)
 <ROAD_1,4,245>: [49648, 49815)
 <ROAD_1,4,246>: [49815, 49982)
 <ROAD_1,4,247>: [49982, 50149)
 <ROAD_1,4,248>: [50149, 50316)
 <ROAD_1,4,249>: [50316, 50481)
 <ROAD_1,4,250>: [50481, 50674)
 <ROAD_1,4,251>: [50674, 50867)
 <ROAD_1,4,252>: [50867, 50926)
 <ROAD_1,4,253>: [50926, 51119)
 <ROAD_1,3,253>: [50926, 51119)
 <ROAD_1,4,254>: [51119, 51312)
 <ROAD_1,3,254>: [51119, 51312)
 <ROAD_1,4,255>: [51312, 51371)
 <ROAD_1,3,255>: [51312, 51371)
 <ROAD_1,3,256>: [51371, 51564)
 <ROAD_1,3,257>: [51564, 51757)
 <ROAD_1,3,258>: [51757, 51816)
 <ROAD_1,3,259>: [51816, 51983)
 <ROAD_1,3,260>: [51983, 52150)
 <ROAD_1,3,261>: [52150, 52317)
 <ROAD_1,3,262>: [52317, 52484)
 <ROAD_1,3,263>: [52484, 52651)
 <ROAD_1,3,264>: [52651, 52816)
 Transaction: DecelerateTransaction: [52816, 55816), {DecreaseSpeedAction: [52816, 55816), {targetSpeed: 4.0}}
 Resource Usage:
 <ROAD_1,3,265>: [52816, 52985)
 <ROAD_1,3,266>: [52985, 53157)
 <ROAD_1,3,267>: [53157, 53332)
 <ROAD_1,3,268>: [53332, 53511)
 <ROAD_1,3,269>: [53511, 53694)
 <ROAD_1,3,270>: [53694, 53881)
 <ROAD_1,3,271>: [53881, 54073)
 <ROAD_1,3,272>: [54073, 54270)
 <ROAD_1,3,273>: [54270, 54472)
 <ROAD_1,3,274>: [54472, 54680)
 <ROAD_1,3,275>: [54680, 54894)
 <ROAD_1,3,276>: [54894, 55114)
 <ROAD_1,3,277>: [55114, 55342)
 <ROAD_1,3,278>: [55342, 55578)
 <ROAD_1,3,279>: [55578, 55816)
 Transaction: ConstantSpeedTransaction: [55816, 57816), {ConstantSpeedAction: [55816, 57816)}
 Resource Usage:
 <ROAD_1,3,280>: [55816, 56066)
 <ROAD_1,3,281>: [56066, 56316)
 <ROAD_1,3,282>: [56316, 56566)
 <ROAD_1,3,283>: [56566, 56816)
 <ROAD_1,3,284>: [56816, 57066)
 <ROAD_1,3,285>: [57066, 57316)
 <ROAD_1,3,286>: [57316, 57566)
 <ROAD_1,3,287>: [57566, 57816)
 Transaction: ChangeToLeftLaneTransaction: [57816, 61817), {ConstantSpeedAction: [57816, 58816), TurnLeftAction: [58816, 60817), {angle: 30}, ConstantSpeedAction: [60817, 61817)}
 Resource Usage:
 <ROAD_1,3,288>: [57816, 58066)

<ROAD_1,3,289>: [58066, 58316]
 <ROAD_1,3,290>: [58316, 58566]
 <ROAD_1,3,291>: [58566, 58816]
 <ROAD_1,3,292>: [58816, 59105]
 <ROAD_1,3,293>: [59105, 59394]
 <ROAD_1,3,294>: [59394, 59483]
 <ROAD_1,3,295>: [59483, 59772]
 <ROAD_1,2,295>: [59483, 59772]
 <ROAD_1,3,296>: [59772, 60061]
 <ROAD_1,2,296>: [59772, 60061]
 <ROAD_1,3,297>: [60061, 60150]
 <ROAD_1,2,297>: [60061, 60150]
 <ROAD_1,2,298>: [60150, 60439]
 <ROAD_1,2,299>: [60439, 60728]
 <ROAD_1,2,300>: [60728, 60817]
 <ROAD_1,2,301>: [60817, 61067]
 <ROAD_1,2,302>: [61067, 61317]
 <ROAD_1,2,303>: [61317, 61567]
 <ROAD_1,2,304>: [61567, 61817]
 Transaction: AccelerateTransaction: [61817, 64817), {IncreaseSpeedAction: [61817, 64817), {targetSpeed: 6.0}}
 Resource Usage:
 <ROAD_1,2,305>: [61817, 62062]
 <ROAD_1,2,306>: [62062, 62298]
 <ROAD_1,2,307>: [62298, 62526]
 <ROAD_1,2,308>: [62526, 62746]
 <ROAD_1,2,309>: [62746, 62960]
 <ROAD_1,2,310>: [62960, 63168]
 <ROAD_1,2,311>: [63168, 63370]
 <ROAD_1,2,312>: [63370, 63567]
 <ROAD_1,2,313>: [63567, 63759]
 <ROAD_1,2,314>: [63759, 63946]
 <ROAD_1,2,315>: [63946, 64129]
 <ROAD_1,2,316>: [64129, 64308]
 <ROAD_1,2,317>: [64308, 64483]
 <ROAD_1,2,318>: [64483, 64655]
 <ROAD_1,2,319>: [64655, 64817]
 Transaction: ConstantSpeedTransaction: [64817, 66817), {ConstantSpeedAction: [64817, 66817)}
 Resource Usage:
 <ROAD_1,2,320>: [64817, 64984]
 <ROAD_1,2,321>: [64984, 65151]
 <ROAD_1,2,322>: [65151, 65318]
 <ROAD_1,2,323>: [65318, 65485]
 <ROAD_1,2,324>: [65485, 65652]
 <ROAD_1,2,325>: [65652, 65819]
 <ROAD_1,2,326>: [65819, 65986]
 <ROAD_1,2,327>: [65986, 66153]
 <ROAD_1,2,328>: [66153, 66320]
 <ROAD_1,2,329>: [66320, 66487]
 <ROAD_1,2,330>: [66487, 66654]
 <ROAD_1,2,331>: [66654, 66817]
 Transaction: ChangeToLeftLaneTransaction: [66817, 70152), {ConstantSpeedAction: [66817, 67817), TurnLeftAction: [67817, 69152), {angle: 30}, ConstantSpeedAction: [69152, 70152)}
 Resource Usage:
 <ROAD_1,2,332>: [66817, 66984]
 <ROAD_1,2,333>: [66984, 67151]
 <ROAD_1,2,334>: [67151, 67318]
 <ROAD_1,2,335>: [67318, 67485]
 <ROAD_1,2,336>: [67485, 67652]
 <ROAD_1,2,337>: [67652, 67817]
 <ROAD_1,2,338>: [67817, 68010]
 <ROAD_1,2,339>: [68010, 68203]
 <ROAD_1,2,340>: [68203, 68262]
 <ROAD_1,2,341>: [68262, 68455]
 <ROAD_1,1,341>: [68262, 68455]
 <ROAD_1,2,342>: [68455, 68648]

```

<ROAD_1,1,342>: [68455, 68648]
<ROAD_1,2,343>: [68648, 68707]
<ROAD_1,1,343>: [68648, 68707]
<ROAD_1,1,344>: [68707, 68900]
<ROAD_1,1,345>: [68900, 69093]
<ROAD_1,1,346>: [69093, 69152]
<ROAD_1,1,347>: [69152, 69319]
<ROAD_1,1,348>: [69319, 69486]
<ROAD_1,1,349>: [69486, 69653]
<ROAD_1,1,350>: [69653, 69820]
<ROAD_1,1,351>: [69820, 69987]
<ROAD_1,1,352>: [69987, 70152]
Transaction: DecelerateTransaction: [70152, 73152), {DecreaseSpeedAction: [70152, 73152), {targetSpeed: 4.0}}
Resource Usage:
<ROAD_1,1,353>: [70152, 70321]
<ROAD_1,1,354>: [70321, 70493]
<ROAD_1,1,355>: [70493, 70668]
<ROAD_1,1,356>: [70668, 70847]
<ROAD_1,1,357>: [70847, 71030]
<ROAD_1,1,358>: [71030, 71217]
<ROAD_1,1,359>: [71217, 71409]
<ROAD_1,1,360>: [71409, 71606]
<ROAD_1,1,361>: [71606, 71808]
<ROAD_1,1,362>: [71808, 72016]
<ROAD_1,1,363>: [72016, 72230]
<ROAD_1,1,364>: [72230, 72450]
<ROAD_1,1,365>: [72450, 72678]
<ROAD_1,1,366>: [72678, 72914]
<ROAD_1,1,367>: [72914, 73152]
Transaction: ConstantSpeedTransaction: [73152, 75152), {ConstantSpeedAction: [73152, 75152)}
Resource Usage:
<ROAD_1,1,368>: [73152, 73402]
<ROAD_1,1,369>: [73402, 73652]
<ROAD_1,1,370>: [73652, 73902]
<ROAD_1,1,371>: [73902, 74152]
<ROAD_1,1,372>: [74152, 74402]
<ROAD_1,1,373>: [74402, 74652]
<ROAD_1,1,374>: [74652, 74902]
<ROAD_1,1,375>: [74902, 75152]
## Transactions in failQueue: 0
# failQueue is empty.
## Transactions in abortQueue
# Number of transactions in abortQueue is: 1
Transaction: ChangeToRightLaneTransaction: [14001, 17336), {ConstantSpeedAction: [14001, 15001), TurnRightAction:
[15001, 16336), {angle: 30}, ConstantSpeedAction: [16336, 17336)}
Resource Usage:
<ROAD_1,2,57>: [14001, 14168]
<ROAD_1,2,58>: [14168, 14335]
<ROAD_1,2,59>: [14335, 14502]
<ROAD_1,2,60>: [14502, 14669]
<ROAD_1,2,61>: [14669, 14836]
<ROAD_1,2,62>: [14836, 15001]
<ROAD_1,2,63>: [15001, 15194]
<ROAD_1,2,64>: [15194, 15387]
<ROAD_1,2,65>: [15387, 15446]
<ROAD_1,2,66>: [15446, 15639]
<ROAD_1,3,66>: [15446, 15639]
<ROAD_1,2,67>: [15639, 15832]
<ROAD_1,3,67>: [15639, 15832]
<ROAD_1,2,68>: [15832, 15891]
<ROAD_1,3,68>: [15832, 15891]
<ROAD_1,3,69>: [15891, 16084]
<ROAD_1,3,70>: [16084, 16277]
<ROAD_1,3,71>: [16277, 16336]
<ROAD_1,3,72>: [16336, 16503]

```

<ROAD_1,3,73>: [16503, 16670)
<ROAD_1,3,74>: [16670, 16837)
<ROAD_1,3,75>: [16837, 17004)
<ROAD_1,3,76>: [17004, 17171)
<ROAD_1,3,77>: [17171, 17336)
STATISTICS
GeneratedTranCount: 26
ConflictedOriginalTranCount: 1
NotAdjustedTranCount: 0
AdjustOnceTranCount: 1
AdjustTwiceTranCount: 0
AlternativeTranCount: 0
AdjustSucceedTranCount: 1
AdjustFailTranCount: 0
AdjustOnceSucceedCount: 1
AdjustOnceFailCount: 0
AdjustTwiceSucceedCount: 0
AdjustTwiceFailCount: 0
AlterSucceedTranCount: 0
AlterFailTranCount: 0
ScheduledTranCount: 26
ExecutedTranCount: 26
CompleteTranCount: 26
IncompleteTranCount: 0
PreemptedTranCount: 0

Appendix B

Sample Simulation Result of CPSNetwork

```
##### CPSNET Simulation Platform
Version: 1.0
Developed by:
Huang Zhu
PhD Candidate
Computing and Information Sciences
Kansas State University
##### Simulation Report of CPSNetwork
Group: GROUP_6, Case: CASE_8
Date of Simulation: Sun Jun 01 02:19:00 CDT 2014
ENV Configuration File: ./CPSNET_ENV_CONFIG.config
CPS Configuration File: ./CPSNET_CPS_CONFIG.config
##### GENREAL INFORMATION
Conflict Resolving: 1
PP Strategy: 2
PC Strategy: 2
Execution Time: [0, 100000)
End Time of the Last Transaction: [0, 76676)
##### STATISTICS
GeneratedTranCount=188
ConflictedOriginalTranCount=14
NotAdjustedTranCount=0
AdjustOnceTranCount=14
AdjustTwiceTranCount=8
AlternativeTranCount=7
AdjustSucceedTranCount=7
AdjustFailTranCount=15
AdjustOnceSucceedCount=6
AdjustOnceFailCount=8
AdjustTwiceSucceedCount=1
AdjustTwiceFailCount=7
AlterSucceedTranCount=3
AlterFailTranCount=4
ScheduledTranCount=184
ExecutedTranCount=184
CompleteTranCount=184
IncompleteTranCount=0
PreemptedTranCount=0
ConflictReservationCount=35
##### MESSAGES
Number of Messages=651
```