RESTCLK: A COMMUNICATION PARADIGM FOR OBSERVATION AND CONTROL OF OBJECT INTERACTION

BY SOURIPRIYA DAS

A dissertation submitted to the
Graduate School—New Brunswick
Rutgers, The State University of New Jersey
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
Graduate Program in Computer Science

Written under the direction of
Professor Chitoor V. Srinivasan

and approved by

_________________________

_________________________

_________________________

_________________________

New Brunswick, New Jersey
January, 1999
© 1999
Souripriya Das
ALL RIGHTS RESERVED
ABSTRACT OF THE DISSERTATION

RESTCLK: A Communication Paradigm for Observation and Control of Object Interaction

by Souripriya Das

Dissertation Director: Professor Chitoor V. Srinivasan

This dissertation introduces a new communication paradigm, RESTCLK, that extends programmatic observation and control (O&C) to the communication network. In a RESTCLK-based distributed application system, any software object with requisite privileges may suspend and resume the flow of data and examine and modify the data in transit over RESTCLK communication networks. Abstractions and protocols used in RESTCLK allow dynamic changes to be performed in RESTCLK communication networks while maintaining transparency with respect to the application objects, that is, without requiring their knowledge or participation.

Support for transparency and dynamicity in RESTCLK networks leads to a unique set of significant features of O&C in RESTCLK including: (1) Objects in a system need not be aware of O&C happening around them; (2) O&C may be performed selectively at any point in a RESTCLK communication network and for any data in transit; and (3) O&C may be recursively performed on O&C itself.
O&C capabilities in RESTCLK lead to several significant benefits, especially in the context of dynamic reconfiguration of distributed application systems: (1) Software objects used in building application systems need not be designed to accommodate future dynamic reconfigurations of application systems; (2) Ability to perform even a priori unplanned dynamic reconfigurations; (3) Reduction in complexity of object source codes through elimination of reconfiguration related coordination logic from source codes; (4) Enhanced reusability of object source codes.

We also discuss use of RESTCLK in several other contexts: In situ testing of new versions of software in their actual target operational environments without jeopardizing correctness of, and with minimal interference to, the services provided by the current versions; Data in transit may be dynamically encrypted and protected as and when need arises over any part of the communication network; Fault management strategies may be dynamically incorporated into an application system.

A proof-of-concept distributed prototype system has been implemented that supports RESTCLK abstractions by providing APIs for use by application system developers. Several well-known applications have been implemented using this prototype to demonstrate some of the benefits of RESTCLK.
Acknowledgements

First and foremost I would like to thank Professor Chitoor Srinivasan for being an excellent advisor and mentor. The many techniques I learned from him for organizing a presentation helped me write a dissertation that is actually readable. I could not have completed this work without his support, encouragement and advice. I would also like to thank my thesis committee members Professor Badrinath, Professor Liviu Iftode, and Dr. Prathima Agrawal for their time, support, and valuable comments.

I am grateful to Professor Ann Yasuhara for keeping a personal touch in our interactions besides her duties as the Graduate Director. Also, I would like to thank Professor Vasek Chvatal for being so professional yet considerate in handling his responsibilities as the Graduate Director.

Graduate secretary Val Rolfe was more like a family member. I am also thankful to Ida Vincler for helping me with some of my administrative needs.

John Scafdi of Electrical Engineering Department has been a good friend. Whenever I needed any help from him, he was there for me. I am thankful to the Department of Electrical Engineering for providing financial support for four years of my graduate study.

I am grateful to Anup Talukdar, Manjunath, Jagannathan, and Pankaj Kumar for taking the time to help me prepare for my defense. I would also like to thank Sanjay Mukherjee for helping me with implementations of RESTCLK and Probal Bhattacharjiya for many useful discussions.
Dedication

I dedicate this dissertation

\textit{to my wife Anindita}

\textit{to her parents and brother}

\textit{to my parents and sisters}
Table of Contents

Abstract ...................................................... ii
Acknowledgements ............................................ iv
Dedication .................................................... v

List of Tables ................................................ xi
List of Figures ............................................... xii

1. Introduction .............................................. 1
   1.1. Problem and Motivation .............................. 1
   1.2. Proposed Solution .................................... 3
   1.3. Objects .............................................. 6
   1.4. The Focus ............................................ 8
   1.5. Principal Contributions .............................. 10
   1.6. Organization ......................................... 12

2. The RESTCLK Paradigm:
   Structure of RESTCLK Communication Network ............ 14
   2.1. Components of a RESTCLK Network .................. 14
   2.2. Communication Between Two Obj-ports ................. 19
   2.3. Indirect Communication in RESTCLK .................. 22
   2.4. Group to Group Communication ......................... 24
   2.5. Negotiation Among Members of a Group ................ 25
   2.6. Machine to Machine Communication ................... 27
2.7. RESTCLK Ports .................................................. 30

3. **RESTCLK Protocols** ........................................ 33
   3.1. Control Signals Used in Protocols .......................... 33
       3.1.1. Communication Control .............................. 36
   3.2. Communication between Agent and Object .................. 36
       3.2.1. Port Sequential Machines ............................ 37
       3.2.2. OPSM States .......................................... 39
       3.2.3. APSM States .......................................... 40
       3.2.4. APSM and OPSM State Transitions .................... 41
   3.3. Group to Group Communication and
       Agreement Protocol .......................................... 47
       3.3.1. CPSM States .......................................... 48
       3.3.2. CPSM Interaction with a Group of OPSMs .............. 48
       3.3.3. The Agreement Protocol in CPSM ..................... 55
   3.4. Other Port to Port RESTCLK Protocols .................... 61

4. **Dynamicty and Transparency** .............................. 67
   4.1. Installation of Objects .................................... 68
   4.2. Creation and Destruction of Obj-ports ..................... 70
   4.3. Installation of Clock Rings ............................... 70
   4.4. Installation of Agents .................................... 71
   4.5. Installation of Watch Rings .............................. 71
       4.5.1. Compatibility of Agent-Port, Obj-Port States .......... 72
       4.5.2. The Tuning Process .................................. 74
   4.6. Transparency .............................................. 79

5. **Significance to Distributed Systems Development** .......... 82
5.1. Input/output Isolation ........................................... 82
5.2. Connection Oriented Architecture .............................. 83
5.3. Security Environment of RESTCLK Network .................... 84
5.4. Communication Transparency ................................... 85
5.5. Active RESTCLK Components .................................. 86
5.6. Unit Buffering and Lossless Communication .................... 87
5.7. Group to Group Communication ................................ 87
5.8. Observation and Control ...................................... 88
5.9. Limitations ...................................................... 88

6. Observation and Control in RESTCLK ........................ 90
   6.1. Observation in RESTCLK ...................................... 90
   6.2. Control in RESTCLK .......................................... 95
   6.3. Applications of RESTCLK O&C .............................. 97
       6.3.1. Halting and Restarting computations .................. 97
       6.3.2. In Situ Testing ........................................ 100
       6.3.3. Fault Management ...................................... 104

7. Security in RESTCLK Networks ................................. 106
   7.1. The RESTCLK Security Layer ................................ 108
   7.2. A Security Management Scheme for RESTCLK ............... 110
       7.2.1. Security Compatibility of Classifications ............ 110
       7.2.2. Validity of RESTCLK Pathways ....................... 111
       7.2.3. Classification of Data ................................ 112
       7.2.4. Time Varying Classifications ........................ 113
       7.2.5. Dynamic Monitoring ................................... 113
   7.3. Concluding Remarks ....................................... 114
8. Implementation .................................................... 115
  8.1. RESTCLK API .................................................. 115
    8.1.1. Synchronization Operations ............................. 118
    8.1.2. Communication Operations .............................. 119
    8.1.3. Configuration Operations ............................... 120
  8.2. Implementation Issues ...................................... 122
    8.2.1. Scalability .............................................. 122
    8.2.2. Communication Efficiency .............................. 124
  8.3. Possible Extensions to RESTCLK ............................ 130
    8.3.1. Pipelining ............................................... 130
    8.3.2. Open Problems .......................................... 136

9. Examples of RESTCLK Applications .............................. 137
  9.1. Dynamic System Replacement in Jacobi Relaxation .......... 139
    9.1.1. Problem Statement ....................................... 139
    9.1.2. RESTCLK Solution to the Reconfiguration Problem .... 144
      Worker Grid Network in RESTCLK ............................ 144
      Worker Algorithm ............................................ 147
      Coordinator Algorithm ...................................... 148
    9.1.3. Comparison of RESTCLK and POLYLITH Solutions ....... 153
      Analysis of RESTCLK solution ............................... 153
  9.2. Dynamic System Evolution .................................. 156
    9.2.1. Problem Statement ....................................... 156
    9.2.2. Dining Philosophers Problem Solution in RESTCLK .... 158
    9.2.3. RESTCLK Solution to Evolving Philosophers ............ 160
      Coordinator Algorithm ...................................... 162
    9.2.4. Comparison with Other Solutions ....................... 165
List of Tables

3.1. Actions Caused by Control Signals in Apsm/Opsm Interaction. . 42
3.2. Actions Caused by Control Signals in Cpsm/Opsm Interaction. . 57
4.1. Compatibility Matrix for Obj-port and Composite Agent-port States. 73
4.2. Action Matrix for Watch Ring Connection . . . . . . . . . . . . 75
8.1. RESTCLK Application Programmer Interface Commands. . . . . 117
9.1. Pseudo-code for Worker Object. . . . . . . . . . . . . . . . . . 149
9.2. Pseudo-Code for Philosopher. . . . . . . . . . . . . . . . . . . 161
9.3. Table for State of New Philosopher & Pending coorDinator Ac-
tivity. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 164
# List of Figures

1.1. The Object Icon ......................................................... 7
2.1. A clock ring with Agents Embedded around its circumference. . . . 15
2.2. The Agent Icon .......................................................... 15
2.3. A composite Agent, the Implementation View of an Agent. ...... 16
2.4. The Watch Ring. ............................................................ 17
2.5. Obj-port to Obj-port Communication Pathway .................... 19
2.6. An Indirect Pathway for Communication between Obj-ports in Dif-
         ferent Clock Rings. .................................................. 23
2.7. Group to Group Communication in RESTCLK. ...................... 24
2.8. Illustrating a RESTCLK Pathway for Conducting Negotiation within 
         a Group. ................................................................. 26
2.9. Hierarchical Organization of Machine to Machine Communication 
         using Representatives. ............................................... 28
2.10. Icon for a RESTCLK Representative. ............................... 29
2.11. Icons Used for Different RESTCLK Ports. .......................... 31
3.1. Activities around a Watch Ring and Agent’s Activity Cycle. . . . 34
3.2. Illustrating the APSM/OPSM Interaction ............................. 38
3.3. Corresponding Activity Cycles of Interacting (APSM, OPSM) Pair. 44
3.4. Corresponding Activity Cycles of Interacting APSM and OPSM 
         Group. ................................................................. 50
3.5. Interacting Psms in Group to Group Communication ................ 55
3.6. The Logic of Agreement Protocol ...................................... 58
3.7. Illustrating Ring-port to Ring-port Communication.  ............. 61
3.8. Illustrating Agent-port to Ring-port Communication.  ............. 62
3.9. Illustrating Obj-port to Watch-port Communication.  ............. 63
3.10. Illustrating Agent-port to Rep-port Communication.  ............. 65
4.1. State Transition Diagram of Illustrating Compatible and Incom-
patible state pairs. ........................................... 78
6.1. Illustrating Different modes of Observation in RESTCLK. ..... 91
6.2. Illustrating the Concept of Immediacy.  ......................... 93
6.3. Illustrating Blocking through Probes.  ........................ 98
6.4. Illustrating An In Situ Testing Scheme in RESTCLK. ........... 102
7.1. RESTCLK Security Architecture.  ............................... 108
8.1. Illustrating the Application System Object Compilation Process. 115
8.2. Illustrating the Application System Architecture. ................ 116
8.3. Illustrating Different Communication Schemes. ................... 126
8.4. Illustrating Mapping of Subsequences of Tokens to Agents on a
Clock Ring. ................................................................ 132
9.1. Flow Chart for Sequential Jacobi Relaxation Technique. ....... 139
9.2. Initial Mapping of Matrix Partitions to a Grid of Workers ....... 140
9.3. Flow-chart of Worker Algorithm. ................................... 141
9.4. Revised Mapping of Matrix Partitions to a New Grid of Workers. 142
9.5. Illustrating Possible Local Iteration Numbers of Workers. ....... 143
9.6. The Initial Configuration of Workers for Solving the Jacobi Relax-
ation Problem. .................................................. 145
9.7. Expanded view of RESTCLK Pathway. .............................. 146
9.8. RESTCLK Network for Dynamic Repartitioning. .................. 150
9.9. Illustrating Worker/Coordinator Interaction in RESTCLK. ....... 153
9.10. Illustrating Worker/Coordinator Interaction in POLYLITH. ....... 154
9.11. Illustrating the Evolving Philosopher Problem . . . . . . . . . . 158
9.12. RESTCLK Networks for Dining Philosophers Problem Solution . . 159
9.13. Illustrating Data Exchanges between Neighboring Philosophers . . 160
Chapter 1
Introduction

1.1 Problem and Motivation

Building flexible dynamically evolving distributed software application systems is a well recognized and generally accepted objective of software designers [25, 41, 19, 43]. Such systems require specialized dynamic management facilities such as,

- **Support for dynamic evolution**: Dynamic evolution can be of two types:
  - *Dynamic reconfiguration*: Here the set of objects (components) constituting a software system and their interconnections may change dynamically, requiring reconfiguration without service interruption.
  - *Dynamic Software Object version changes*: Here the existing version of software objects used in a running system may have to be replaced with their new versions without service interruption [41, 40, 13, 18].

- *In situ Version Testing*: New versions of software objects have to be tested in their actual target operational environment without jeopardizing correctness of current service and also with minimal interference to the service.

- **Fault Management**: Recovery from faults may require dynamic reconfigurations to be designed and performed at the time faults occur [28] during an application system’s lifetime.
- **Dynamic Security**: Application system should support a dynamic changes in its security environments.

Presently providing support for the above kinds of management facilities would require that software objects be designed to accommodate all anticipated management needs for the target application system. Current design techniques would require that software objects in an application system have embedded in them code for their collaboration in all the anticipated management tasks [25, 19]. Since it is impossible to anticipate all future management needs at design time, it becomes clearly impossible to design flexible systems which can respond to all contingencies which may arise during the lifetime of an application system.

As a consequence one faces the following problems:

- Source code complexity of software objects increases as a result of the need to embed within them code for management and collaboration logic.

- This causes objects to become specialized to specific application contexts in which they are actually used.

- This leads to reduced reusability of objects, because an object often cannot be reused in a context where needed object functionality remains the same but collaboration requirements are different.

- Application systems built with such objects may not be able to respond to unanticipated management needs.

These problems usually result in application systems with only limited capabilities for dynamic evolution.

Software systems by their very nature are prone to changes\(^1\) Building change management tools for large software systems has always been a significant part of

\(^1\)Indeed, it has been argued that introduction of the computing system itself a stimulus for change [29].
their design and implementation. Building flexible software systems, in the sense that required change management needs have been already provided for, has been an important issue in software engineering [24, 25].

Principal proposals that identified collaboration requirements and developed mechanisms to program needed collaborations came from systems such as, Conic, Podus, Polylith and PCL [25, 41, 10, 44]. All proposed solutions assumed that to achieve management flexibility in a system one must incorporate collaboration related code into individual software objects. Recent work in this area [12, 31], thus focused mainly on language and programming environment aspects that might simplify specification of the needed collaboration code.

Existing approaches to the design of dynamically evolving distributed software systems thus severely limit evolutionary capabilities of resultant systems.

Objective: Our objective is to develop environments for design and implementation of distributed application systems which can in principle accommodate practically unlimited dynamic evolution.

1.2 Proposed Solution

To realize this objective we advocate here a fundamental shift in paradigm. It calls for total elimination of collaboration related codes from object source codes. Collaboration needed for reconfiguration will be achieved by guidance provided to objects by applications specific management objects, called managers. These managers need not be designed at the time an application system itself was designed and implemented. They can be designed and incorporated into an application system at any time during its lifetime whenever the need arises.

Guidance provided by managers to each object will be determined by giving access to managers to a carefully selected finite subset of states of the object called disclosure states. At any time an object will be in one these disclosure
states or in transition from one to another. Whenever the need arises a manager may request the object to divulge one or more of its disclosure states and the object will oblige immediately or as soon as it reaches a consistent point at which it could divulge the requested state(s). Thus disclosure states of objects will be accessible to managers.

Selecting the disclosure states of a software object is a far simpler task than identifying required collaboration logic and incorporating the code for handling such collaboration into the object. It may be noted, designer of a software object need not be concerned with management tasks in which disclosure states might be used; disclosure states of an object can be determined completely from its internal characteristics without reference to network contexts in which it might be used.

In addition to incorporation of facilities to divulge disclosure states, we shall require that objects should contain facilities which make it possible to initialize their respective states. Also, we shall require that consistent states of an application system should be definable in terms of the disclosure states of its constituent objects. Most importantly, starting from an initial state it should be possible to guide objects through their state transitions by selectively feeding appropriate inputs to them at the right times. This will make it possible to bring application systems to consistent states by feeding its constituent objects the right inputs at the right times.

Thus in principle, the behavior of each object should be reducible to that of one or more finite state machines whose states would be the disclosure states. We believe this to be a reasonable requirement that is not difficult to satisfy. Incidentally, it also imposes a useful software object design discipline.²

Clearly, if source codes of objects in a system are independent of their network

²It may be noted that these are not new requirements. We have here simply made explicit assumptions which have been implicit in all current approaches to dynamic reconfiguration.
contexts, and if network contexts themselves can be dynamically changed with no participation required from the objects, then it should be possible to perform dynamic reconfigurations even if they had not been a priori planned for. Also, any given network context of any object in a system may be dynamically and transparently changed (i.e. with no service interruption and no object collaboration), whenever such changes are feasible without destroying application system functionality and consistency.

A well designed system will then consist of objects which satisfy all state disclosure requirements for all management tasks that it might potentially encounter during its lifetime, whether such tasks were anticipated or not at the time of its design. Managers needed for any such reconfiguration task may themselves be then designed and implemented when need arises during the lifetime of an application system. Such systems will implicitly possess the capability for practically unrestricted dynamic evolution during their lifetimes, limited only by the sets of disclosure states designed into their constituent software objects.

The principal claim made here is the following:

Claim: By enabling well designed distributed application system objects to exercise the right kinds of programmatic control over data flowing through communication networks, it is possible to achieve the objective stated above.

Two critical properties to be satisfied by such programmatic control are: dynam- icity and transparency. By dynam- icity we mean that such control should be exercisable without service interruption at any time during the lifetime of a system. By transparency we mean that exercise of such control should not require knowledge or collaboration of application objects whose interactions are being subjected to such control.

However, at the time this work stated there were no concepts available as to
• What the objectives of such programmatic control of communication should be;

• What abstractions one might use to specify such control;

• What communication architectures would be appropriate;

• What benefits one might accrue by its use?

RESTCLK paradigm answers all these questions. Why and how it does, is the subject of this dissertation. We will begin our discussion with the concept of objects used in RESTCLK.

1.3 Objects

A large distributed software system consists of a set of software components, hereinafter called objects, and their interconnections. Objects in the system use the interconnections to exchange input/output (I/O) data among themselves. In RESTCLK we assume, every object will come with a capability to support a certain number of object ports [2, 11, 16], hereinafter referred to as obj-ports. Three of these obj-ports are assumed here to exist in every object. They are designated for special use [16, 42].

(i) State port: This will be used to get and set the internal state of an object,

(ii) Instrumentation Port: This will be used to make measurements on object performance and

(iii) Emergency Port: This will be used to control an object when emergency malfunctions occur.

---

3The general assumptions described here are not necessary for use of RESTCLK communication system. We present here one possible way of organizing an application system, that is consistent with corba [16] recommendations, in order to fully exploit RESTCLK capabilities.
Figure 1.1: The Object Icon

We will discuss later, in Chapter 6, details on how these ports may be used by an application system using the RESTCLK paradigm.

In addition to these ports an object may have one or more additional I/O ports as well. Figure 1.1 illustrates the object-icon we use in this dissertation to refer to such software objects. The small circular items embedded on the circumference of the object-icon represent its obj-ports: The figure shows an object with the designated ports mentioned above and several I/O ports. Often we will use object-icons showing only their I/O ports; the designated ports are assumed to be always there. During its lifetime an object may dynamically create and destroy any of its ports, except the designated ports. We assume throughout, when an object is installed it will always come with all of its initially needed ports.

We also assume that each object will service its obj-ports using schedules that are driven only by its own internal activity requirements. Thus, the times at which an obj-port is polled to determine service requirements will be fixed only by its parent object. Also, after polling an obj-port a parent object may not immediately perform all computations needed to complete servicing it. It may suspend computations needed for servicing an obj-port at any time and resume service later at its own choosing. In general it is advantageous to design objects with this capability to suspend and later resume obj-port servicing. We assume throughout, objects will have this capability. RESTCLK will not impose
on objects any predetermined object port service requirements.\textsuperscript{4}

As is well known [39], each such object of a distributed system may be separately designed, programmed, compiled and tested. This is referred to as \textit{programming in the small}. Defining network configurations in which objects reside and their communication interfaces is referred to as \textit{programming in the large} or \textit{configuration programming} [23]. The subject matter of this dissertation is configuration programming.

A set of objects working together will implement a distributed system's functionality by communicating and coordinating their activities. Support given by a software development environment for communication and synchronization is often critical to the design and development of distributed systems with requisite flexibility.

\subsection{1.4 The Focus}

The focus of RESTCLK is communication among objects, not the objects themselves. RESTCLK provides a communication environment for distributed systems that is application independent. Each object will receive inputs from and send outputs to other objects only \textit{via} its obj-ports. We will refer to the set of objects with which a given object, say X, may communicate as the \textit{network context} of X.

We assume throughout that for each object its designers would characterize its behavior in terms of a carefully selected set of \textit{disclosure states}. We believe, developing design criteria and guidance for how this might be done is itself a legitimate subject for future research. More experience with the use of RESTCLK is needed in order to do this meaningfully. Examples discussed in this dissertation are not enough to comment meaningfully on this subject.

Each object in a given network context will have to necessarily satisfy two basic

\textsuperscript{4}This is the \textit{Input/Output isolation} property.
requirements: One is I/O-\textit{functionality} and the other is communication functionality, C-\textit{functionality}. I/O-functionality will specify relationship which should hold between inputs and outputs associated with each obj-port of an object. C-functionality in a given network context will specify such things as interface conventions, who may exchange data with whom, protocols for sending and receiving data, buffering requirements and synchronization of senders and receivers to make sure each will be listening to other during communication sessions.

A central issue in the design and implementation of distributed systems is the relative independence with which these two functionalities may be realized [16]. Total independence would imply that for any obj-port its I/O-functionality will be preserved independent of changes in its C-functionality. Thus protocols used by obj-ports may change, network contexts may change, synchronization conditions may change and through all such changes the I/O-functionality will be preserved.\footnote{It may be noted that the property of \textit{independence} as presented here is stronger than the input/output isolation property. Independence not only requires input/output isolation relative to internal operations of an object, but also requires an invariant object behavior modulo changes in network contexts and protocols.}

If total \textit{independence} is realized then one may design and implement objects to contain only the minimal amount of code needed to realize its I/O-functionality at each one of its obj-ports. Such objects may then be used in a variety of network contexts by appropriately redefining its C-functionality in each context without in any way changing the internal codes of the objects. This is the ideal one seeks in flexibility. It is a necessary feature for building and using reusable object libraries.

Unfortunately this ideal is currently not realizable in practice, because I/O- and C-functionalities get inevitably intertwined in object implementations [25]. Thus at present objects designed and implemented for use in one network context will often become unusable when the context changes.

RESTCLK paradigm attempts to approximate the ideal mentioned above
in the context of a particular kind of active communication network, called
RESTCLK-network. The architecture of this active network makes it possible
to separate out to a large extent implementation issues of I/O-functionalities
from the realization of its associated C-functionalities.

Two other significant features of RESTCLK are \textit{dynamica} and \textit{transparency}:
Software systems using the RESTCLK organization may be dynamically reconfig-
ured with minimal or no interruption of services. We refer to this as \textit{dynamica}.
System functionality and consistency can be preserved during such reconfigu-
ations without any collaboration from the objects themselves. Objects need not
even be aware of changes occurring around them. We refer to this latter property
as \textit{transparency}.

\subsection*{1.5 Principal Contributions}

In conventional programming once data leaves a program it goes outside the scope
of programmatic control of application programs. It usually goes into the domain
of a \textit{communication} subsystem. Programmatic control of data in transit is at
present not a well defined concept. As mentioned earlier, it not known what
the purpose of such programmatic control should be, and what abstractions are
needed to specify and manipulate such control.

In this work we postulate that the purpose of acquiring programmatic control
of data in transit should be \textit{Observation and Control (O&C)}: \textit{Observation} here
implies a capability to access and examine data in transit at any time and at
any point along their pathways by any application \textit{object} with requisite privileges.
\textit{Control} implies a capability to suspend, resume and redirect flow of data in transit
along communication pathways, and a capability to modify them as necessary, at
the discretion of any application \textit{object} with requisite privileges.
We introduce three fundamental abstractions: *ports*, *agents* and *rings*. We show how communication networks that support distributed computations may be defined, constructed and dynamically modified using these abstractions (Chapters 2 and 4). We refer to such networks as RESTCLK networks. Communication among *agents* and *ports* in RESTCLK networks is coordinated and synchronized by a set of new protocols, called RESTCLK protocols (Chapter 3).

We identify *independence*, *dynamicity* and *transparency* as the three important properties which any such data communication network should support in order to provide the necessary O&C capabilities. We show how the protocols used in RESTCLK networks satisfy these properties. Many interesting and useful benefits may be derived from such O&C capabilities. We demonstrate some of them by focusing on the problem of *dynamic reconfiguration*. This has been an active area of research and development for more than ten years.

We show how RESTCLK networks may be used to design and enforce dynamic protection and security measures for data in transit, at any point and at any time during their migration through a communication network (Chapter 7). We also show how RESTCLK networks may be used for *in situ* testing of new versions of software in their actual operational environment without interfering with services being provided by their earlier versions (Chapter 6).

We demonstrate here, using well known examples [25, 19], how RESTCLK facilities may be used for dynamically reconfiguring distributed systems, even when such reconfigurations were not *a priori* planned for and application *objects* were not designed to accommodate such reconfigurations. We argue that this leads

---

6It may be noted, the concept of *ports* and *agents* have been used in other systems. However, the concept, structure and operation of *ports* and *agents* in RESTCLK are quite different from those in other systems. The concept of *rings* is unique to RESTCLK.

7See references cited under Related Work, Section 9.4.

8We are not aware of any other system which supports dynamic security and *in situ* testing facilities of the kind supported by RESTCLK.
to two significant consequences, namely, reduction in complexity and enhanced reusability of object codes. Also, arguably this strengthens the ability of an application system to respond to unanticipated failures [28].

A proof-of-concept prototype has been implemented, that supports RESTCLK abstractions by providing an Application Programming Interface (API) which can be used by application system programmers. Several scalability and efficiency issues related to RESTCLK are also discussed (Chapter 8).

1.6 Organization

This dissertation is organized as follows: The RESTCLK paradigm is discussed in Chapter 2. This introduces components used in a RESTCLK network and kinds of network structures that may be built using the components. This is followed by a detailed discussion in Chapter 3 of RESTCLK protocols which are used to coordinate communication among network components. The ensuing chapter, Chapter 4, presents how network components may be dynamically created and destroyed in RESTCLK without in any way interfering with computations being performed by objects in a distributed system. This is the transparency property referred to earlier. Significance of this RESTCLK communication architecture to distributed systems development is discussed in the next chapter, Chapter 5. The ensuing chapter, Chapter 6 presents some important methods of observation and control that RESTCLK communication system allows, to dynamically observe, reorganize and otherwise modify a distributed system without interfering with services provided by the system and without requiring the system itself to be designed a priori to accept such changes.

In the next chapter, Chapter 7 we present the basic facilities provided in RESTCLK that allows a distributed system developer to extend the security environment of an application system to the RESTCLK communication network. It
also presents certain novel ways of dynamic security management and monitoring facilities that are unique to RESTCLK.

The next chapter, Chapter 8, presents a brief discussion of the prototype RESTCLK implementation, certain scalability and efficiency issues, discusses extensions to RESTCLK which can significantly improve its efficiency and introduces some open problems. Of course, there will be a price to pay for the benefits that RESTCLK offers. One may expect that an application system using RESTCLK communication would not work as efficiently as one using plain sockets for communication. We have not done any experimentation to estimate or assess cost penalties that RESTCLK may impose. It is beyond the scope of this work. Our objective here is only to demonstrate a proof of concept. However, based on the analysis presented in Section 8.2 one may claim that this cost would not only be reasonable but probably be outweighed by the gained benefits. Also, RESTCLK is a new paradigm and surely with experience more efficient implementations of RESTCLK will come.

The ensuing chapter, Chapter 9, presents a few examples of RESTCLK applications. This chapter also compares RESTCLK with POLYLITH and CONIC, which have also proposed organizing principles for communication in distributed systems. The examples discussed in Chapter 9 were used to demonstrate applications and uses of RESTCLK facilities and its benefits. The thesis concludes with Chapter 10 which summarizes the main contributions made by this dissertation.
Chapter 2

The RESTCLK Paradigm:
Structure of RESTCLK Communication Network

2.1 Components of a RESTCLK Network

In the RESTCLK paradigm its communication net is a set of circular communication pathways, called *rings*, with active *agents* embedded in them, as shown in the Figure 2.1. Data may travel around a ring only in one direction; we will usually assume clockwise flow of data around rings. Each agent on a ring will be responsible to collect data from its clients, package the data in appropriate ways and pass them on, together with the necessary *control signals*, to its next agent on the ring. Also, each agent may belong to exactly one ring. Thus, rings in RESTCLK may not intersect with each other. The ring to which an agent belongs is called the *parent ring* of the agent.

RESTCLK networks will contain two kinds of rings: *Clock Rings* and *Watch Rings*. The structure of these two kinds of rings is described below.

**Clock Ring:** Figure 2.1 shows a clock ring. Agents on a clock ring are called *clock agents*. Clock rings are homogeneous: they may contain only clock agents. Figure 2.2 illustrates the expanded view of the triangular icon we will use to represent clock agents. As shown in the figure, each clock-agent will come with several *ports* attached to it. The port at the top of the triangle in the figure is called the *ring port*. This is the port that is used to attach the clock agent to
Figure 2.1: A clock ring with Agents Embedded around its circumference.

Figure 2.2: The Agent Icon.
its parent clock ring, as shown in Figure 2.1. A clock-agent on a clock ring will receive control signals from its previous agent on its parent ring, and send them to its next agent, via its ring port.

Ports at the bottom of a clock-agent, shown in Figure 2.2 are called agent-ports. A clock agent may have more than one agent-port attached to it. A clock agent will receive (send) data and control signals from (to) its clients via its agent-ports. Each agent-port of a clock agent will be connected to a client that is being served via a watch ring, as described later below.

It should be noted that this view of an agent as consisting of one ring port and multiple agent-ports is useful only for the purpose of explaining the functions performed by clock agents and establishing their properties. The implementation view of agents is shown in Figure 2.3: Here the agent is shown with one ring port at its top and a large composite agent-port at its bottom connected to multiple watch rings. The composite agent-port at the bottom will implement the functions jointly performed by all the individual agent-ports shown in Figure 2.2. This is the way clock agents have been implemented in the current RESTCLK implementation and this is the way we will hereafter represent clock agents in all our diagrams.

Watch Ring: Figure 2.4 illustrates the elliptical ring we use to represent
watch rings: they have a heterogeneous structure. Each watch ring will be attached to one agent-port and one obj-port: The agent-port is at left in the figure and the obj-port is at right. Other ports attached to the watch ring in the figure are called watch-ports. We will shortly discuss their function.

The side of a watch ring along which data and control signals travel from the agent-port (at left) to the obj-port (at right) is called its \textit{delivery side}. The side along which data and control signals travel from the obj-port back to the agent-port is called its \textit{dispatch side}. The essential differences between these two sides arise because of the nature of control signals that they carry: As we shall later see, signals sent by the agent-port to the obj-port along the delivery side will be different from the signals sent by the obj-port back to the agent-port along the dispatch side. As shown in the figure, on the delivery side the watch-port is the same as the ring-port but on the dispatch side the watch-port is different, as explained later.

As shown in Figure 2.4 one may attach \textit{watch agents} to either sides of a watch ring \textit{via} their respective watch-ports. A watch agent on the delivery side may be used to intercept data being sent to an obj-port and one on the dispatch side may

![Figure 2.4: The Watch Ring.](image_url)
be used to intercept output data coming out of an obj-port. As shown in the figure, each watch agent will distribute intercepted data to its client objects via its own composite agent-port. There is no intrinsic limit to the number of watch agents one may attach to a watch ring.

It may be noted that clock and watch rings are abstractions (tabular data structures) which hold slots in which agents may be inserted. They provide pathways in the sense agents on such ring tables will normally transmit data from one to its next one cyclically (last agent on the table communicating back with the first) during their activities. Circularity of rings reflect the fact that all communication in RESTCLK use reply semantics. Normally a ring will contain only one or two agents. Multiple agents on rings shown in diagrams here will appear only on special circumstances in an application system when the system seeks to observe and control its own activities. The observer agents and their associated watch rings may all be dynamically installed whenever need arises for observation and control. As soon as the observation and control task is completed observer agents and associated watch rings may all be immediately removed.

These are the RESTCLK components that one will use for communication between objects in one machine. When communication involves multiple machines then data exchange between machines will occur in the RESTCLK network using a third kind of agent, called representative. We will discuss the network structure in which representatives are used in a subsequent section.

The manner in which the above RESTCLK components may be used to exchange data from one obj-port, $p_1$, to another, $p_2$, is discussed in the next subsection.
2.2 Communication Between Two Obj-ports

Figure 2.5 illustrates a RESTCLK network which may be used to establish a two way communication path between two obj-ports $p_1$ and $p_2$. The figure shows two agents $A_1$ and $A_2$ mediating communication between $p_1$ and $p_2$: $p_1$ is connected to $A_1$ via the watch ring $W_1$, $A_1$ is connected to its next agent $A_2$ via a segment of the clock ring $C_1$ and $A_2$ is connected to $p_2$ via the watch ring $W_2$.

A communication session between agent $A_1$ and obj-port $p_1$ will begin with $A_1$ sending some data to $p_1$ via the delivery side of the watch ring $W_1$. Then $A_1$ will wait to receive the response from $p_1$ to the data it just delivered. $A_1$ will receive $p_1$'s response via the dispatch side of $W_1$. This data will be gathered and packaged in the appropriate manner by $A_1$ and then forwarded on to its next
agent $A_2$ via the segment of the clock ring $C_1$ that connects $A_1$ to $A_2$. We will refer to such a cycle of data exchange between a tuned (agent-port, obj-port) pair as a *communication session*.

Clearly, an (agent-port, obj-port) may be tuned to each other if and only if they are connected to each other by a watch ring and each has authority to send and receive data from the other.

When $A_2$ receives data sent by $A_1$ along the clock ring segment, $A_2$ will begin its own communication session with $p_2$ via $W_2$. In the first half of this session $A_2$ will forward the data it received from $A_1$ to $p_2$ and in the second half, receive $p_2$'s response, if any, to this data. The communication session will end when $A_2$ forwards the data it has gathered from $p_2$ to its next agent, say $A_3$, in the clock ring $C_1$.

At this point it should be noted that the parent object of $p_2$, the object $O_2$ in the figure, could do anything it liked with the data it received from $A_2$: $O_2$ could merely forward it back to $A_2$ via $p_2$; it could examine it and then forward it back; it could change it as it liked before forwarding it back; or $O_2$ could send back to $A_2$ an entirely different set of data which constituted its own response to the data it received. Thus, all agents in the cyclic sequence,

$$\langle A_1, A_2, A_3, \ldots, A_k, \ldots, A_1 \rangle$$

on the clock ring $C_1$, starting and ending at $A_1$ going clockwise around the ring, will have an opportunity to have a communication session with their own respective client objects, when they intercept the data migrating around the ring. Ultimately, when data completes a full cycle of migration around the clock ring $C_1$ it will get back to the agent $A_1$. Thus for every complete round of migration around a clock ring, data migrating around the ring will participate in as many *communication sessions* as there are agents on that ring.

This is the basic obj-port to obj-port communication paradigm in RESTCLK.
A complete communication pathway between two distinct obj-ports may contain a minimum of two agents, both attached to the same clock ring, and two watch rings. We will later discuss an exception to this rule in the context of communication among members of a group.

We will say that watch rings are used to tune obj-ports to agents that serve them, and clock rings tune clockwise adjacent agents on the same ring. Management objects of an application system are responsible for establishing such communication pathways between obj-ports in the system. The obj-ports participating in data exchange may not themselves know the identities of entities with whom they are communicating.

RESTCLK protocols discussed in the next section will guarantee that gathering of data from senders and forwarding data to recipients will always occur at right times, and in each communication session an agent and its client(s) will always be listening to each other throughout the session.

In the current implementation of RESTCLK it is true that at any time only one bag (collection) of data may go around a ring and only one agent on the clock ring will have an active communication session. It is useful to note that this is not a requirement for RESTCLK networks. As discussed in Section 8.3 it is possible to implement RESTCLK to hold as many active communication sessions around a clock ring as there are agents on the ring: thus, a clock ring with \( n \) agents may have up to \( n \) concurrent communication sessions. In this case more than one agent on a clock ring may be active at any given time.

RESTCLK allows a management object to dynamically create new agents on any segment that lies between adjacent pairs of agents on the ring. By introducing agents on a clock ring segment one may obtain a capability to examine and/or change the data flowing along that segment of the ring.

The important feature that makes RESTCLK paradigm useful is that agents and rings may all be dynamically created and installed without having to interfere
in any manner with ongoing computations being performed by the objects in an application system. The manner in which this is done is discussed in Chapter 4. Part of the reason for this is that all RESTCLK induced modifications would occur only on the RESTCLK communication net. An object may not even be aware that its data is being accessed by others. This puts a significant burden on RESTCLK to make sure that application system security will never be violated. We discuss in Chapter 7 details on how an application system programmer may use the RESTCLK facilities to extend the security environment of an application system to the entire RESTCLK communication network.

2.3 Indirect Communication in RESTCLK

As mentioned earlier, RESTCLK provides facilities for dynamically tuning and detuning (agent-port, obj-port) pairs. Using this facility one may thus create temporary pathways between obj-ports tuned to agents in different clock rings. Thus if the obj-port $p_1$ in Figure 2.5 had to communicate with another port $p_4$ tuned to agent $A_4$ on another clock ring $C_2$, RESTCLK may be used to detune existing pathways and set up a new pathway between $p_1$ and $p_4$. Once data transfer between $p_1$ and $p_4$ is completed the original network configuration may be restored.

An alternative way for $p_1$ to communicate with $p_4$ might be by using an application specific data transfer object, which has the property that it would always transfer data received from one of its obj-ports to another one that it owns. This is illustrated in Figure 2.6. The important point to note here is that RESTCLK does not limit in anyway pairs of obj-ports which may communicate with each other.
Figure 2.6: An Indirect Pathway for Communication between Obj-ports in Different Clock Rings.
2.4 Group to Group Communication

Figure 2.5 illustrates a special case where data is exchanged between just one pair of obj-ports. The general situation in RESTCLK is illustrated in Figure 2.7. Here a group of objects

\[ G_1 = \{O_1, \ldots, O_i, \ldots, O_n\} \]

is communicating with another group

\[ G_2 = \{B_1, \ldots, B_i, \ldots, B_m\}. \]

The obj-ports \(O_{p_1}, \ldots, O_{p_i}, \ldots, O_{p_n}\) of the objects in group \(G_1\) are all tuned to the same agent \(A_1\) in the clock ring \(C_1\). The obj-ports \(O_{q_1}, \ldots, O_{q_i}, \ldots, O_{q_m}\) of the objects in group \(G_2\) are similarly tuned to agent \(A_3\). Communication between these two groups will be mediated by the two agents \(A_1\) and \(A_3\). The other two agents on the clock ring, shown in the figure, might be observer agents monitoring this communication. They are, of course, optional.
The clock agent $A_1$ here will be responsible to gather all data sent to it by all obj-ports in group $G_1$ and put them in a *bag*. When bagging is complete $A_1$ will forward the bag to $A_3$ *via* the observer agent $A_2$, if one existed. When $A_3$ receives this bag (or the bag possibly modified by the clients of $A_2$) it will deliver it simultaneously to all obj-ports in group $G_2$. We refer to this as *simultaneous data delivery*, since data from all objects in group $G_1$ are delivered together at one time. When agent $A_3$ completes gathering data sent to it by members of the group $G_2$ it will forward the data back to $A_1$, again *via* agent $A_4$, if one existed.

Memberships of objects in groups $G_1$ and $G_2$ may both change dynamically without interrupting any of the communication sessions. Such changes in membership may be implemented at any time by detuning existing members from their clock-agents or by tuning in new members to them, as necessary.

### 2.5 Negotiation Among Members of a Group

It is not uncommon that objects belonging to a group find a need to communicate among themselves either in the context of a negotiation in which all members should participate, or in the context of some kind of committee work. Such communication may be organized in RESTCLK as illustrated in Figure 2.8. The figure shows a *manager* at the top. This manager might monitor the negotiation without necessarily participating in it, or might be involved in resolving conflicts among members of the group. The presence of such a manager is, of course, optional. In the arrangement shown, negotiation will proceed in steps.

If no manager is present, then the clock ring will contain only one agent. In this case, the agent will collect data from every member of the group into a *bag* in each communication session, and in the next session distribute this *bag* of data to every member. This will constitute one step of the negotiation. Thus, at the beginning of each step all members of the group will receive at one time
Figure 2.8: Illustrating a RESTCLK Pathway for Conducting Negotiation within a Group.
all messages sent by members of the group in the previous step.\(^1\) Next step of negotiation will begin with the responses generated by each member of the group for the data they received at the end of the previous step. This process will continue until an agreement is reached.

This organization will guarantee that in each negotiation step every member of the group will have access to the same negotiation history. Thus mutual knowledge about negotiation history that participants have may all be made identical. This feature is used in OMSOFT [43] to implement a *tetherless negotiation* facility for applications development.

If a manager is present then \(A_1\) will forward its data bag to the manager, instead of distributing it back to the members. After examining the messages, the manager will forward it back to \(A_1\) who will then distribute it to all members of the group. The manager will thus have an opportunity to add his/her own comments, or modify the data bag in other manners. Each member will then receive the data forwarded by the manager at the end of each negotiation step. Again they will all have the same mutual knowledge at every step of the negotiation. Members of such a group may, of course, be distributed over different machines in a network.

The RESTCLK structure that makes machine to machine communication possible is discussed below.

2.6 Machine to Machine Communication

It is possible that not all members of a communicating group, which are tuned to the same clock agent, reside in the same computer. Members residing in different machines will get tuned to the same agent using the hierarchical organization shown in Figure 2.9. Here obj-ports \(P = \{p_1, p_2, \ldots, p_n\}\) that reside in different computers are all tuned to the same agent, \(A_1\). In this case tuning is accomplished

\(^1\)This is the *simultaneous delivery* property satisfied by RESTCLK agents, mentioned earlier.
Figure 2.9: Hierarchical Organization of Machine to Machine Communication using Representatives.
through the use of *representatives*. The agent $A_1$ will have a representative in each machine in which members of the set $P$ reside. Of course, an agent will not need such a representative if all members of $P$ are in its own machine. Details of the icon used to denote representatives is shown in Figure 2.10. The essential difference between an agent and its representative is the following: Whereas an agent connects to its parent clock ring through its ring port, a representative connects to the composite agent-port of the agent it represents, *via* some inter-machine communication network that is external to RESTCLK. A representative will use a special port, called the *rep-port*, to connect to this external network. We will discuss later the differences between rep-ports, agent-ports and ring-ports. The icon used to depict a representative is the same as the icon for an agent but for the letter $R$ inscribed in it. It is shown in Figure 2.10. Figure 2.9 shows a different kind of watch ring with broken lines that connect agent ports to a rep-ports. This is a network connection, that is external to RESTCLK. Patterns of control and data flow over these connections are similar to those over a regular watch-ring connection, as explained later. However there is a difference: Whereas one can arbitrarily add observer watch-agents to a regular watch-ring, it is not possible to do this to a agent-port/rep-port connection, since the connection is provided by an external network.
For the members of $P$ who reside in the same machine as the representative, the representative will perform the agent’s tasks on that machine. Thus in each machine the representative of an agent will be responsible to coordinate collection and distribution of data to obj-ports in $P$ that reside in that machine. The representative will receive data from parent agent in a different machine via the inter-machine communication network. Thus, representative $R_1$ in Figure 2.9 will receive data from $A_1$ and distribute this data to all the clients in its machine. It will send back to $A_1$ data collected from all its clients, via the inter-machine network.

As shown in the figure this kind of interaction between an agent and its representative can be extended to several machines in a network through a hierarchical organization. Thus the interaction that occurs between an (agent, representative) pair can also occur between a (representative, representative) pair. In such an organization a communication session between an agent and its client group will end only when all data from all representatives in a hierarchy have reached the agent at the root.

We will later discuss in Section 8.2.1 the advantages of this kind of hierarchical organization. We will conclude this introduction to RESTCLK network structure with a summary of the different kinds of ports used in the network.

2.7 RESTCLK Ports

There are basically four different kinds of ports used in RESTCLK. These are illustrated in Figure 2.11. The differences between the various ports arise because of the control signals they send and receive. There are basically three kinds of control signals: advance, release and trigger. The nature of these control signals and activities initiated by them are discussed in the next chapter. For our purposes here we will merely note the classification of ports based on control
Each port is represented by two concentric circles. The outer circle represents the signal that is sent out by a port and the inner circle represents the signal that is received, as indicated in the diagrams in Figure 2.11. The colors of these circles determine the kinds of signals a port sends or receives: White color denotes advance signals and dark color denotes release and/or trigger signals. Following this convention we get four different kinds of icons to represent the four different kinds of RESTCLK ports, as illustrated in the Figure 2.11. These are explained below:

- **A/A-Port**: This is the icon for ring-ports and watch-ports on the delivery side of a watch ring. A/A-Ports receive and send only advance signals. Thus the inner and outer circles are both white.

- **A/R-Port** This is the icon used for obj-ports and rep-ports. These ports receive advance signals and send out release/trigger signals. Thus the inner circle is white and the outer circle is dark.

- **R/R-Port** This is the icon used for watch-ports on the dispatch side of a watch ring. These ports receive and send release/trigger signals only. Thus both the inner and outer circles are dark.

- **R/A-Port** This is the icon for agent-ports. These ports receive release/trigger signals and send out advance signals. Thus the inner circle is dark and the
outer circle is white.

Reader may note the following (please refer to port diagrams in Figure 2.11 as the following is being read):

If port X communicates with port Y then the outer color of port X should match with the inner color of port Y. If X and Y communicate with each other then they should have complementary inner and outer colors, respectively. Thus an A/R-port (obj-port) may have two way communication with an R/A-port (agent-port) and vice versa. This is what happens over a watch ring. However, an A/A-port (ring-port) can never communicate with an R/R-port (watch-port on dispatch side). An A/A-port may communicate only with another A/A-port. This happens on clock rings and may happen on delivery side of watch rings. R/R-port may communicate with R/A-port (agent-port), but the reverse communication can never happen. Similarly, an A/R-port (obj-port) may communicate with an R/R-port while the reverse communication can never occur. Also similarly, an R/R-port may communicate with another R/R-port.

We use the iconic port representations introduced here throughout this dissertation.

The principal aspects of RESTCLK that make the above described RESTCLK paradigm function are the protocols used for communication between the various types of RESTCLK ports. We will refer to these protocols as RESTCLK protocols. They regulate and synchronize all communication occurring in RESTCLK networks, allowing each object in an application system to proceed with its computations unhindered by communication exigencies. RESTCLK protocols coordinate beginnings and endings of communication sessions that occur while data migrate around rings. These are discussed in detail in Chapter 3.
Chapter 3
RESTCLK Protocols

We assume throughout that data transported around a RESTCLK network will at every point in the network reside in some designated memory. RESTCLK will have the responsibility to assign to objects, agents and representatives their respective read/write privileges for these memories. This model holds, of course, only for shared memory communication. When data travels across a distributed network from one machine to another, RESTCLK will logically view the network line itself as memory.

3.1 Control Signals Used in Protocols

All protocols in RESTCLK are defined in terms of four binary control signals: advance signals, release signals and two kinds of trigger signals, pre-release trigger and post-release trigger. These control signals are used to synchronize communication between ports. They enable agents and their representatives to gather and distribute data at the right times. These control signals will mark the beginnings and endings of activities in communication sessions between ports.

One may think of each port as having a port sequential machine (PSM), whose state transitions are caused by these control signals. We will refer to the port associated with a PSM as the parent port of the PSM. In each state the PSM associated with a port will cause one or more of the following tasks to be performed:

(a) Send to or receive from another port data and/or control signals.
Figure 3.1: Activities around a Watch Ring and Agent’s Activity Cycle.

(b) Go to a designated next state.

(c) Each state transition is associated with generation of specified output signals.

We will use interacting sequential machine model to describe the protocols. The roles played by the control signals in psm interaction are briefly described below and illustrated in Figure 3.1. The figure shows the schematic of connections between an agent-port, $X$ of an agent $A_i$ and an obj-port $Y$, via a watch ring whose control and data pathways are shown separately. Data travels around the watch ring from $X$ to $Y$ and back to $X$, through the designated memories $M_A$ and $M_Y$ of the agent $A_i$ and port $Y$, respectively. On the delivery side of the ring only the advance signal travels from agent $A_i$ to $Y$. On the dispatch side both release and trigger signals travel from $Y$ to $A_i$. The pulse at the bottom of the figure shows the activity cycle associated with a tuned pair of ports, $(X, Y)$ in
a communication session. The activity cycle consists of an active period followed by an idle period. Data communication between $X$ and $Y$ will occur only during the active period. Data will be exchanged via the designated memories as shown in the figure. We will first describe below the various control signals and their purposes. The nature of activities that occur during an activity cycle is discussed in Section 3.2.4 below.

**Control Signals:**

- **Advance**: Port $X$ may send an *advance* signal to port $Y$ to indicate that data in memory $M_A$ is ready to be read by the parent of $Y$. As shown in the activity cycle pulse in Figure 3.1 receipt of *advance* will mark the beginning of the active period of a new activity cycle.

- **Release**: $Y$ may send a *release* signal to $X$ to indicate that output data has already been written in $M_Y$ and is ready to be read by $X$. Always this will mark the end of the active period in the current activity cycle of $Y$.

- **Pre-release and Post-release Triggers**: $Y$ may send a *pre-* or *post-release trigger*\(^1\) to $X$ in order to indicate that $Y$ would like the data in memory $M_Y$ to be forwarded as soon as possible. Forwarding of data by agent $A_1$ will occur only when an *a priori* agreed upon coordination condition is satisfied.

If $Y$ is the only port tuned to $X$ then coordination condition will be satisfied when $X$ receives both *release* and *trigger* from $Y$. If at any time multiple ports $Y_1, Y_2, \ldots, Y_n$ are tuned to $X$, then $X$ will be a composite agent-port. In this case data from all the individual private memories, $M_{Y_1}, M_{Y_2}, \ldots, M_{Y_n}$, will be gathered together by $X$ and forwarded if and only if the following coordination condition is satisfied: *Release* signals have been received from every port $Y_1, Y_2, \ldots, Y_n$.

\(^1\)When it is not important to distinguish between *pre-* and *post-* release triggers we will simply refer them as *triggers.*
... $Y_n$ and a trigger signal has been received from at least one of them. We will refer to this condition as the agreement protocol.

State transitions that occur in port sequential machines and outputs generated by them are discussed in the next section.

### 3.1.1 Communication Control

In any communication there is one that initiates and another that listens. At any point in time the entity that can initiate communication will be said to have control of communication at that point of time. A communication session may be viewed as a sequence of transfers of communication control between the parties that are communicating. In every clock ring, at any point of time, communication control over the clock ring will reside with one agent on the ring. As data migrates around the ring this control will transfer clockwise from one agent to its next agent.

We will assume throughout that each clock ring will have a designated memory associated with it. This memory is denoted in Figure 3.1 by $M_A$.\(^2\) At any point in time only the agent on the clock ring who has communication control, will have read/write access to this designated memory, $M_A$. An obj-port $Y$ that has communication control over a watch ring will have read access to its agent’s designated memory $M_A$ and write access to its own private memory $M_Y$.

### 3.2 Communication between Agent and Object

In the following, we will consider the special case where communication is between an agent-port, $X$, and its only client obj-port, $Y$,\(^3\) and the agent-port $X$

---

\(^2\)In the current implementation we designate two memory units for each ring. At any point in time one of these two will be used for reading only and the other for writing only.

\(^3\)This is to be contrasted with group communication where a composite agent-port communicates with multiple client obj-ports concurrently. This case is discussed in Section 3.3.
is communicating directly via a watch ring with the obj-port, i.e., there are no intervening watch-ports on the delivery or the dispatch side of the watch ring. Also, there is only one watch ring connected to the agent. Initially communication control over the watch ring will reside with the agent-port X.

### 3.2.1 Port Sequential Machines

Here, communication will occur as a result of the interaction between the Agent-Port Sequential Machine, \textit{APSM}, of \textit{X} and the Obj-Port Sequential Machine, \textit{OPSM}, of \textit{Y}. This is illustrated in Figure 3.2. The top part of the figure shows the two \textit{PSMs} with their respective inputs and outputs. The same inputs and outputs are shown also in the bottom diagram of the figure. This diagram also shows the states of the port sequential machines, their transitions for the various inputs, and outputs they generate during the transitions. The inputs received by these machines will cause state transitions and outputs, as described below.

The \textit{APSM} receives one input, ‘a’, from its parent agent. This is the \textit{advance} input. The only state in the \textit{APSM} that has a transition for this input is the state \textit{A}_{TR}; the transition is to the state \textit{A}_{Tr}. This should be interpreted as indicating that the only state in which the \textit{APSM} will receive input ‘a’ is its state \textit{A}_{TR}. During the transition, as shown in the figure, the \textit{APSM} will emit output ‘a’ to the \textit{OPSM} it is tuned to. Other transitions shown in the figure should be similarly interpreted. We will soon discuss interpretations associated with states shown in the figure.

As shown in the figure, the \textit{APSM} sends out ‘agreement’ signal to its parent agent. This output is emitted during transitions shown in the figure. The only output the \textit{APSM} sends to the \textit{OPSM} is the output ‘a’ mentioned above.

The \textit{OPSM} sends three outputs to the \textit{APSM}: ‘t’, ‘t-pre’ and ‘t-post’ and one output to its parent object, namely ‘a.’ The transitions during which these outputs are generated and the inputs that cause these transitions are shown in
Figure 3.2: Illustrating the APSM/OPSM Interaction.
the Opsm shown in the figure. The Opsm receives two inputs, ‘r’ and ‘t’, from
its parent. The sending and receiving of these control signals will mark the
beginnings and endings of communication control among ports as described below.

Each PSM in the figure has four states. The states of these PSM have been
labeled as follows: Lower case subscripts, as in \( O_{tr} \), refers to the state in which
Opsm has sent neither a \textit{trigger} nor a \textit{release}. Similarly, state \( A_{tr} \) will refer to
the state in which the Apsm has received neither a \textit{trigger} nor a \textit{release}. We use
capital subscripts, as in \( O_{TR} \), to refer to the state in which Opsm has already
sent both a \textit{trigger} and a \textit{release}. Similarly, state \( A_{TR} \) will refer to the state of the
Apsm in which it has received both a \textit{trigger} and a \textit{release}. The subscripts “\( T_r \)”
and “\( t_R \)” are used to specify the other two possibilities. These are summarized
below:

3.2.2 Opsm States

- \( O_{tr} \): Neither triggered nor released. In this state Opsm will have control of
communication around the watch ring. Thus, this is an \textit{active} state.

- \( O_{TR} \): Triggered but not released. Opsm has emitted a pre-release trigger,
but the release itself has not yet been emitted. Opsm will still be in control
of communication around the watch ring. This is also an \textit{active} state.

- \( O_{tR} \): Not triggered but released. Opsm has emitted a release, but no trigger
has been emitted. Opsm may later optionally send a post-release trigger.
In this state the Opsm is not in control of communication around the watch
ring. This is an \textit{idle} state.

- \( O_{TR} \): Both release and trigger have been emitted. Again control of commu-
nication around the watch ring is not with the Opsm. This is also an \textit{idle}
state.
The two states $O_{IR}$ and $O_{TR}$, both of which indicate that a \textit{release} signal has already been sent, are the idle states. These are the dark colored states in Figure 3.2. As mentioned above, an \textsc{OpSm} that is idle will not have control of communication. No reading or writing of data may take place after a \textit{release} has been sent. As mentioned above, always an \textsc{OpSm} will receive \textit{advance} signal from the \textsc{ApSm} it is tuned to, while the \textsc{OpSm} is in its idle state $O_{TR}$. We will later see in Section 3.3 on group to group communication, why an \textsc{OpSm} can be in either one of its idles states, $O_{TR}$ or $O_{IR}$, at the time it receives this \textit{advance}.

### 3.2.3 Apsm States

- $A_{tr}$: Neither \textit{trigger} nor \textit{release} has been received. In this state, control of communication over the parent clock ring of $A_i$ is with the agent $A_i$, but control of communication over the watch ring is not.

- $A_{Tr}$: \textit{Trigger} has been received but not \textit{release}: The trigger received here would be a \textit{pre-release trigger}. Control of communication over the watch and clock rings will be the same as in the previous case.

- $A_{ir}$: A \textit{release} has been received but not a \textit{trigger}. \textsc{ApSm} will later receive a \textit{post-release trigger}. In this state control of communication over the watch ring and clock ring is with the agent.

- $A_{TR}$: Both \textit{release} and \textit{trigger} have been received. Here communication control over watch ring will reside with agent and control over the clock ring will move to next agent on the ring.

Thus, while an \textsc{ApSm} is in one of its active states $A_{tr}$, $A_{Tr}$ or $A_{ir}$, its parent agent $A_i$ will be in control of communication over its clock ring. In the idle state $A_{TR}$ this is not guaranteed. Again, the dark colored states in Figure 3.2 are the idle states.
We begin our discussion here assuming that $A_i$ is not in control of communication over its parent clock ring. At this point the APSM associated with the agent-port of $A_i$ will be in its idle state $A_{TR}$. This APSM will be in control of communication over the watch ring.

### 3.2.4 APSM and OPSM State Transitions

**Initial States:** Initially both OPSM and APSM will be in their respective idle states $O_{TR}$ and $A_{TR}$. Communication control over the watch ring will be with the APSM of the agent-port $A_i$, and communication control over the clock ring will be with agent $A_{i-1}$. When $A_i$ is in its idle state, as mentioned earlier, it will not have read/write access to the designated memory $M_A$.

**Transitions from Idle to Active State:** $A_i$ will acquire control of communication over its clock ring when it receives an *advance* from the previous agent $A_{i-1}$ in its parent clock ring. After obtaining control of communication from $A_{i-1}$, $A_i$ will send an *advance* to its agent-port $X$, i.e., to the APSM shown in Figure 3.2. On receipt of this advance the APSM will move from its idle state $A_{TR}$ to its active state $A_{tr}$ and send an *advance* signal to the obj-port $Y$ via the delivery side of the watch ring. This output is shown in Figure 3.2 as the output emanating from the transition from $A_{TR}$ to $A_{tr}$. This state transition is shown in row ADV1 of Table 3.1. This transition will begin a communication session between the APSM and the OPSM tuned to it, as described below.

*Advance* signals may thus be viewed as wake up signals that cause their recipients to take control of communication.

The various PSM state transitions and interactions between APSM and OPSM are discussed in the next subsection.

When $Y$ receives this advance its OPSM will get read-access to the designated memory $M_A$. Also, control of communication around the watch ring will transfer
<table>
<thead>
<tr>
<th>Row#</th>
<th>FSM</th>
<th>Current State</th>
<th>Input</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADV1</td>
<td>APSM</td>
<td>$A_{TR}$</td>
<td>Advance from Parent Agent</td>
<td>$A_{tr}$</td>
<td>Advance to OPSM</td>
</tr>
<tr>
<td>ADV3</td>
<td>OPSM</td>
<td>$O_{TR}$</td>
<td>Advance from APSM</td>
<td>$O_{tr}$</td>
<td>Advance to Parent Object</td>
</tr>
<tr>
<td>ADV4</td>
<td>OPSM</td>
<td>$O_{tr}$</td>
<td>Advance from APSM</td>
<td>$O_{tr}$</td>
<td>Advance to Parent Object</td>
</tr>
<tr>
<td>TRG1</td>
<td>OPSM</td>
<td>$O_{TR}$</td>
<td>Trigger from Parent Object</td>
<td>$O_{TR}$</td>
<td>Pre-release Trigger to APSM</td>
</tr>
<tr>
<td>TRG2</td>
<td>OPSM</td>
<td>$O_{tr}$</td>
<td>Trigger from Parent Object</td>
<td>$O_{TR}$</td>
<td>Post-release Trigger to APSM</td>
</tr>
<tr>
<td>TRG3</td>
<td>APSM</td>
<td>$A_{TR}$</td>
<td>Pre-release Trigger from OPSM</td>
<td>$A_{TR}$</td>
<td>No Action</td>
</tr>
<tr>
<td>TRG4</td>
<td>APSM</td>
<td>$A_{TR}$</td>
<td>Pre-release Trigger from APSM</td>
<td>$A_{TR}$</td>
<td>‘agreement’ to Parent Agent</td>
</tr>
<tr>
<td>RLS1</td>
<td>OPSM</td>
<td>$O_{TR}$</td>
<td>Release from Parent Object</td>
<td>$O_{tr}$</td>
<td>Release to APSM</td>
</tr>
<tr>
<td>RLS2</td>
<td>OPSM</td>
<td>$O_{TR}$</td>
<td>Release from Parent Object</td>
<td>$O_{TR}$</td>
<td>Release to APSM</td>
</tr>
<tr>
<td>RLS3</td>
<td>APSM</td>
<td>$A_{TR}$</td>
<td>Release from OPSM</td>
<td>$A_{tr}$</td>
<td>No Action</td>
</tr>
<tr>
<td>RLS4</td>
<td>APSM</td>
<td>$A_{TR}$</td>
<td>Release from OPSM</td>
<td>$A_{TR}$</td>
<td>‘agreement’ to Parent Agent</td>
</tr>
</tbody>
</table>

Table 3.1: Actions Caused by Control Signals in APSM/OPSM Interaction.

to $Y$. The state transition corresponding to this acquisition of communication control by the OPSM is shown in Figure 3.2 as the OPSM transition from state $O_{TR}$ to state $O_{tr}$ with label ‘a’, and row ADV3 of Table 3.1.\footnote{The transition and output shown in row ADV4 of the table may occur at the time of establishing a watch ring connection to a newly created obj-port, as described in Chapter 4. This may also occur at any time in group communications as described in Section 3.3.} Also, as shown in the figure, at this point the OPSM will emit an advance to its parent object.

It should be noted, at the time $Y$ thus takes control of communication around the watch ring, $A_i$ will still be in control of communication around its parent object.
clock ring, i.e. $A_i$ will still have read/write access to its parent ring’s designated memory.

When the parent object of the OPSM notices that the OPSM is in state $O_{tr}$, it will know that there is input data available in $M_A$ for its use. The parent object may then read this input data and use it. Also, while the OPSM is in its active state $O_{tr}$, the parent object may write its outputs to the obj-port’s designated memory (see Figure 3.1).

**Transitions from Active to Idle State:** While in active state $O_{tr}$, at the request of its parent object, the OPSM may send a *pre-release trigger* to APSM and make a transition to state $O_{Tr}$ (see row TRG1 in Table 3.1). On receiving this pre-release trigger the APSM will make a transition to its $A_{Tr}$ state, as shown in row TRG3. These transitions and associated outputs are also shown in Figure 3.2. No transfer of communication control is involved here.

While OPSM is in one of its active states $O_{tr}$ or $O_{Tr}$, at the request of its parent object, the OPSM may send a *release* to the APSM and make a transition to one of the idle states $O_{tR}$ or $O_{TR}$, respectively. These are shown in Figure 3.2 and also in rows RLS1 and RLS2 of Table 3.1. On receipt of this release from OPSM the APSM will make a corresponding transition to either $A_{tR}$ or $A_{TR}$ state, respectively. These are also shown in the figure and in rows RLS3 and RLS4. These transitions will cause communication control around the watch ring to transfer from the OPSM back to the APSM.

An OPSM will send a *post-release trigger* if its parent object requests it to send a *trigger after a release* has been already sent. In this case the OPSM will make a transition from $O_{tR}$ to $O_{TR}$, as shown in the figure and in row TRG2 of Table 3.1. No transfer of communication control will occur here.

On receiving this post-release trigger the APSM will make its transition from $A_{tR}$ to $A_{TR}$ as shown in the figure and in row TRG4 of the table. The idle period is the interval during which a PSM is in any of its idle states: $O_{tR}$ or $O_{TR}$ for
Figure 3.3: Corresponding Activity Cycles of Interacting (APSM, OPSM) Pair.

OPS M and $A_{TR}$ for APS M. When an OPS M is in idle state it will not be in control of any communication. When an APS M is in idle state it will always be in control of communication over its associated watch ring. But its parent agent may or may not be in control of communication over the parent agent’s clock ring.

Activities in a communication session that take place during such an (APSM, OPSM) interaction may be visualized in terms of activity cycles shown in Figure 3.3. As mentioned earlier, each activity cycle will consist of an active period followed by an idle period. As shown in the figure the active period is the interval during which a psm is in any of its active states: $O_{tr}$ or $O_{TR}$ for OPSM, and $A_{tr}$, $A_{TR}$ or $A_{tr}$ for APSM. When an APSM is in active state its parent agent will be in control of communication over its parent clock ring. While an OPSM is in active state its parent port will be in control of communication over the associated
watch ring.

Pulse [A] in the figure is the activity pulse of the agent-port. Pulses [O₁] and [O₂] are the two possible ways in which activity might end in an APSM/OPSM interaction. Active period begins in [A] when the APSM sends an advance signal to the OPSM at time T₀. This signal is received by the OPSM after some delay. When the OPSM receives this signal it becomes active by going to state Oᵣ in both pulses [O₁] and [O₂].

In pulse [O₁] a pre-release trigger is sent by the OPSM at time Tᵣ. This is received by the APSM after some delay. This causes the APSM to move to state Aᵣ. Activity of the OPSM ends in [O₁] when it sends the release at time Tᵣ and moves to state Oᵢ. This release is received by the APSM after some delay at time Tᵣ, which is the time when activity of the agent-port ends.

Pulse [O₂] illustrates the other possibility. Here release is sent in [O₂] at time Tᵣ before a trigger is sent. Soon after sending this release the OPSM moves to state Oᵢ. When the APSM receives this after some delay it moves to state Aᵢ as well. At a later time Tᵣ the OPSM in [O₂] sends a post-release trigger and moves to state Oᵢ. This trigger is received by the APSM after some delay at time Tᵣ, when it moves to state Aᵢ and ends its activity.

The idle periods of the APSM will end when the next advance is sent at time T₁ and the idle period of the OPSM will end when this advance is received at time Tᵢ, as shown in Figure 3.3. In both cases termination of the active period of the APSM occurs when it has received both a trigger and a release. The figure shows the corresponding states of the APSM and the OPSM in both cases.

As has been explained above, an active period of an OPSM will always begin with an advance and end with a release. Its idle period will begin with a release and end with the next advance. The active period of an APSM will end only when both release and trigger have been received.

It may be noted, an obj-port will leave its active state only when it sends a
release. If it does not send a release then it will continue to be in active state and hold communication control over its associated watch ring. Similarly, an agent-port will lose communication control over its associated watch ring only when it sends an advance to the obj-port tuned to it. If it does not send this advance then it will continue to hold communication control over the watch ring.5

The Inclusion Relationship: We refer to the following relationship between the active and idle periods of an APSM and its corresponding OPSM as the inclusion relationship:

- The active period of an APSM will always include the active period of its corresponding OPSM.
  
  In other words, when an OPSM holds communication control over its watch ring, the APSM tuned to it will always be listening to the OPSM.

- The idle period of an OPSM will always include the idle period of its corresponding APSM.
  
  In other words, when the APSM is holding communication control over the watch ring, the OPSM attached to it will be always be listening to the APSM.

The fact that this inclusion relationship will always hold is of critical importance to all communication in the RESTCLK networks: Any time an agent-port is communicating with an obj-port the entity that is not holding communication control will always be listening to the one that is holding it.

Changes to a RESTCLK network are always made in a manner that would preserve this relationship. As will be shown in Chapter 4,

5These may be detected, when necessary, by using ‘time-outs.’
it is this feature that makes it possible to realize \textit{dynamicity with transparency} in RESTCLK networks.

In the next section we show how this inclusion relationship is preserved in group to group communication, where one agent may service multiple obj-ports in a group. The use of \textit{agreement protocol} will become evident in this case.

### 3.3 Group to Group Communication and Agreement Protocol

A schematic diagram of group to group communication appears in Figure 2.7. We will here focus on the interaction that takes place between the agent $A_1$, shown in Figure 2.7, and the group of obj-ports,

$$G_1 = \{Op_i \mid 1 \leq i \leq n\}.$$ 

We will use the label, $Op_i$, to denote both an obj-port and its OPSM. In this discussion we first assume that all obj-ports of the group would reside in a single computer. The case where the obj-ports in a group reside in different computers is discussed later in Section 3.4 in the context of agent-port/rep-port protocol. Data will migrate from agent $A_1$ to members of the group $G_1$ through the designated memory $M_A$ of the agent; it will migrate from the individual obj-ports $Op_i$ back to $A$ through their respective designated memories, say $M_i$ for $i = 1, 2, \ldots n$. These designated memories are not shown in the figure.

Interpretation of states of a Composite agent Port Sequential Machine, CPSM, with multiple watch rings is different from the interpretation of states of an APSM with a single watch ring discussed earlier. The CPSM states are described in the next subsection.
3.3.1 CPSM States

We will use $C_{\alpha\beta}$ for $\alpha \in \{t, T\}$ and $\beta \in \{r, R\}$ to denote the four possible states of a CPSM. A composite agent may, of course, have more than one watch ring connected to it. The states of the composite agent-port during an activity cycle are defined as follows:

1. $C_{TR}$: The composite port will be in state $C_{TR}$ iff it has received a trigger from at least one watch ring and received release signals from all the watch rings, i.e., the agreement protocol for the agent is satisfied.

2. $C_{tR}$: The composite port will be in state $C_{tR}$ iff it has received no triggers from any of the watch rings and received releases from all watch rings connected to its composite agent port.

3. $C_{Tr}$: The composite port will be in state $C_{Tr}$ iff it has received a trigger from at least one watch ring and not received release from at least one watch ring.

4. $C_{tr}$: The composite port will be in state $C_{tr}$ iff it has received no triggers from any of the watch rings and has not received release from at least one watch ring.

The agent of a composite port will have have communication control over its parent clock ring in all states enumerated above except the state $C_{TR}$. The composite port itself will have communication control over all watch rings connected to it in state $C_{TR}$ and $C_{tR}$. In other states it may not have control over all watch rings connected to it.

3.3.2 CPSM Interaction with a Group of OPSMs.

Interaction between the group $G_1$ and agent $A_1$ may be visualized in terms of the inclusion relationship between composite activity cycle associated with the
agent \( A_1 \) and the individual activity cycles of each obj-port in group \( G_1 \). This is illustrated in Figure 3.4 and explained below.

The activity cycle \([A] \) at the bottom of the figure is the cycle associated with the agent \( A_1 \). Cycles \([X], [Y] \) and \([Z] \) at the top of the figure illustrate the three possible distinct scenarios that may occur in a CPSM/OPSM interaction: \([X] \) shows the case where pre-release trigger (i.e., trigger before release) occurs; \([Y] \) shows the case where post-release trigger (trigger after release) occurs, but this trigger is still received at \([A] \) before the end of its active period and \([Z] \) shows the case where the post-release trigger reaches \([A] \) after \([A] \) has terminated its active period.\(^6\) Some OPSMs in the group \( G_1 \) will have activity cycles of type \([X] \), others will have cycles of type \([Y] \) and yet others of type \([Z] \). The distribution of these types of activity cycles among the obj-ports in \( G_1 \) will, of course, vary from one communication session to another.

In Figure 3.4 the time axis is shown at the bottom. At time \( A_n^0 \) the advance signal is sent by the composite APSM of agent \( A_1 \) to all obj-ports in group \( G_1 \). At this same time the CPSM moves from its idle state \( C_{TR} \) to its active state, \( C_{tr} \). The activity cycle of pulse \([A] \) depicts this state transition as a transition from level 0 to level 1 in the pulse. This transition marks the beginning of a new activity cycle in the CPSM. One should notice here that at the time an agent \( A_1 \) sends an advance the composite APSM will always be in its idle state \( C_{TR} \). However, this does not require that all the OPSMs in group \( G_1 \) serviced by the agent \( A_1 \) should be in the corresponding idle state \( O_{TR} \). As shown in the figure, in pulse \([Z] \), some could be in state \( O_{TR} \) and others in state \( O_{tr} \). This is one of

\(^6\)Actually it is possible that in a communication session an obj-port in such a group never sends a trigger. In such a case only release would have been sent. For such a contingency to arise there must be prior agreement among members of the group, which allows certain members to skip triggers in certain activity cycles. Another variation of this theme is allowing delayed sending of triggers with maximum allowable delays, where delays exceeding the maximum will be cause for initiating fault management. Occurrences of exceptional situations like these will be application dependent. In the discussion presented here we will assume that such exceptions will not arise.
Figure 3.4: Corresponding Activity Cycles of Interacting APSM and OPSM Group.
the consequences of the agreement protocol discussed below.

The advance signal $A^0_a$ sent by the CPSM will be received by the different obj-ports $O_{pi} \in G_1$ at different times, because of varying transit delays. In the figure, the thin lines showing the migration of the advance signal to pulses $[X]$, $[Y]$ and $[Z]$, depict these delays. As soon as an OPSM in $G_1$ receives the signal $A^0_a$ it will move from its idle state, which may be $O_{TR}$ or $O_{tR}$, to its active state $O_{tr}$ as shown in the figure. While in this active state the OPSM and its parent object will read and use inputs and may also write outputs.

Any time during this input/output process, as shown in diagram $[X]$ of Figure 3.4, the OPSM $O_{pi}$ associated with the pulse $[X]$ may send a pre-release trigger to the CPSM. This will cause the OPSM to immediately transfer to its next state $O_{Tr}$, as shown in pulse $[X]$ in the figure. This signal reaches pulse $[A]$ of the CPSM after some delay. One may think of this pre-release trigger as signifying that the parent object of the OPSM would like the agent $A_1$ to forward its output data as soon as agreement protocol is satisfied. Satisfaction of agreement protocol will indicate that all members in $G_1$ have declared completion of all of their respective tasks in the current communication session.

It is, of course, quite possible that at the time one OPSM sends its pre-release trigger another OPSM communicating with the same agent might have already sent both its trigger and release. It is also, of course, possible that there are some OPSMs which have sent neither release nor trigger. The agent will keep track of the triggers and releases received from each OPSM that is tuned to it in a communication session.

In the diagram shown in Figure 3.4 the first trigger received by the CPSM at time $A_t$ is the pre-release trigger sent by $[X]$, as shown in pulse $[A]$. This is the time when the APSM would register receipt of a trigger. All trigger signals (whether pre- or post-) received subsequent to the first trigger received by the CPSM will be effectively ignored, i.e., would cause no state changes in the CPSM.
Figure 3.4 also shows the CPSM receiving releases sent by [X], [Y] and [Z] after varying delays. The time $A_r$ shown in the figure should be construed as being the time at which the CPSM had received releases from all its clients. In fact at time $A_r$ in this figure the CPSM had not only received releases from all of its clients but had also received a trigger from one of its clients. Hence, as shown in the diagram, it terminates its active period.

In practice, the order in which the OPSMs in $G_1$ send their respective triggers and releases will, of course, depend on the application and details of computations involved in any one communication session. The agreement protocol will guarantee that no communication session would end before all communicants tuned to an agent have explicitly indicated the end of their participation in the session. They indicate this by sending release signals.

In any communication session, all post-release triggers received by a CPSM after satisfaction of the agreement protocol will be ignored by the CPSM. Diagram [Y] illustrates one such possible case, where a communicant in $G_1$ sends its post-release trigger after the agent had already detected satisfaction of the agreement protocol.

It should be added at this point that any member in the group $G_1$ may decide to leave the group at any point in time, even in the middle of a communication session, by simply detuning itself from the agent $A_1$. The agent will enforce the agreement protocol only among the OPSMs that are tuned to it as of the time at which the agreement protocol itself is satisfied. Thus, if an OPSM detunes itself before the end of the agreement protocol the agent $A_1$ will no longer expect to receive either release or trigger from that agent.\footnote{This makes it possible to use timeouts in a communication session as a strategy to detune malfunctioning members of a group when necessary.}

In any communication session, an agent $A_1$ will destroy or change data in its designated memory $M_A$ only after it had sensed satisfaction of
the agreement protocol.

One may add new members to a group by tuning them to agent $A_1$ in the middle of a communication session. As long as these new members joined the group before the satisfaction of the agreement protocol,

(a) input data in the agent’s designated memory will still be available for their scrutiny, and

(b) the agreement protocol will be satisfied only if the new members also send all of their respective release and/or trigger signals.

The general inclusion relationship that guarantees synchronization of data exchange in a communication session between an agent and a group is described below.

**The General Inclusion Relationship:** As shown in Figure 3.4, for each OPsm in the communicating group $G_1$ its active period in an activity cycle will end and its idle period will begin only when it had sent a release. The time point at which this happens to the different obj-ports in $G_1$ will, of course, be different. Also, the time points at which agent $A_1$ receives these releases will be delayed by varying amounts. These are shown in Figure 3.4 by the dotted lines that start at time points on pulses $[X]$ and $[Y]$ and end at time points on the composite pulse $[A]$. The time at which the active period of the composite pulse ends will be the time at which the agent $A_1$ senses completion of the agreement protocol. The receipt of the next advance signal by the agent $A_1$, namely signal $A^1_{sa}$ in Figure 3.4, will mark the end of $A_1$’s idle period and the beginning of the next communication session.

Thus as shown in the figure, it will be always true that in any communication session the active and idle periods of the composite pulse $[A]$ will always satisfy the inclusion relationship:
(i) The active periods of the members of the group $G_1$ will always be inside the active period of the agent $A_1$.

Thus every obj-port tuned to a composite agent-port is guaranteed that as long as it is holding communication control over their associated watch rings, the agent-port they are tuned to will always be listening to them.

(ii) The idle period of the agent $A_1$ will always be inside the idle period of every member in $G_1$.

Thus the agent $A$ is guaranteed that as long as it is holding communication control over its associated watch rings, all obj-ports tuned to it will be listening to it.

Properties (i) and (ii) will always hold between every pair of tuned OPSM and CPSM in RESTCLK no matter how many members participate in communication groups. This will guarantee that every time any agent $A_1$ sends an advance signal to obj-ports tuned to it all the obj-ports will be listening to it. Also, any time an obj-port sends a release or trigger to its agent, the agent will be listening to it. Thus in every communication session both the agent and all obj-ports tuned to it will be listening to each other at appropriate times. This will guarantee lossless data exchange. Also, this guarantees that agents and obj-ports would need only unit buffers: This is because new data will be delivered to a group only when the current data has been already read and used.

These are the behaviors that should be realized in an agent to group communication by a CPSM. It may be noted that all post-release triggers shown in Figure 3.4 occur before the beginning of the next activity cycle, i.e. before time $A_1^a$ in the figure. This may not, however, always happen. It is quite possible that a post-release trigger sent by a client might reach the CPSM after the time $A_1^a$. When this happens, one should make sure that the CPSM would recognize this late post-release trigger as belonging to the previous activity cycle and hence
ignore it. This may happen because of a race condition between the client sending the late post-release trigger and the next advance being sent by the agent at time $A^n$. RESTCLK solves the potential problems this race condition may cause, by extending the APSM discussed earlier to include within its operation the logic of agreement protocol, as discussed in the next subsection.

### 3.3.3 The Agreement Protocol in CPSM

The Composite Port Sequential Machine, CPSM, is shown in Figure 3.5. It has two parts to it: The top part is its state diagram. Where as the APSM shown in Figure 3.2 received its inputs directly from its client OPSM, here inputs from
the Opsms go first to the agreement protocol box as shown in the figure. The outputs from this box are used in the CPSM state diagram. We shall later discuss the agreement protocol logic and how this logic resolves problems caused by the race condition, mentioned earlier.

Inputs to CPSM state diagram shown in Figure 3.5 are:

- ‘\( \land r \)’: This stands for \( (r_1 \land r_2 \land \ldots \land r_n) \), the conjunction of release signals produced by the client Opsms.

- ‘\( \lor t \)’: This stands for \( (t_1 \lor t_2 \lor \ldots \lor t_n) \), the disjunction of triggers, which are derived by the agreement protocol logic, described below, from the pre-release and post-release triggers produced by the clients in the group.

- ‘a’: This is the advance input received from the parent agent of the CPSM.

But for the above changes in inputs the CPSM state diagram is identical to the APSM state diagram discussed earlier. State transitions that occur in the CPSM/OPSM interaction are summarized in Table 3.2. Let us now consider the role played by the agreement protocol logic. It may be noted that the OPSM transitions and actions are the same in both Table 3.1 and 3.2. This is significant because the role played by an OPSM does not change whether communication is between an agent and a group of obj-ports or between an agent and just one obj-port.

The logic of the agreement protocol is shown in Figure 3.6. Inputs to the agreement protocol box are the triggers and releases produced by the client Opsms. These inputs are first saved in a set of input registers, as shown in the figure. As mentioned earlier, satisfaction of the agreement protocol will signal the end of the active period of an activity cycle. As shown in the figure, the 'agreement' signal that represents this satisfaction itself is used to clear the registers. Thus at the end of the active period of each activity cycle all registers in the agreement
<table>
<thead>
<tr>
<th>Row#</th>
<th>PSM</th>
<th>Current State</th>
<th>Input</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADV1</td>
<td>CPSM</td>
<td>C&lt;sub&gt;TR&lt;/sub&gt;</td>
<td>Advance from Parent Agent</td>
<td>C&lt;sub&gt;tr&lt;/sub&gt;</td>
<td>Advance to Opsm</td>
</tr>
<tr>
<td>ADV3</td>
<td>OPSM</td>
<td>O&lt;sub&gt;TR&lt;/sub&gt;</td>
<td>Advance from CPSM</td>
<td>O&lt;sub&gt;tr&lt;/sub&gt;</td>
<td>Advance to Parent Object</td>
</tr>
<tr>
<td>ADV4</td>
<td>OPSM</td>
<td>O&lt;sub&gt;IR&lt;/sub&gt;</td>
<td>Advance from CPSM</td>
<td>O&lt;sub&gt;tr&lt;/sub&gt;</td>
<td>Advance to Parent Object</td>
</tr>
<tr>
<td>TRG1</td>
<td>OPSM</td>
<td>O&lt;sub&gt;tr&lt;/sub&gt;</td>
<td>Trigger from Parent Object</td>
<td>O&lt;sub&gt;TR&lt;/sub&gt;</td>
<td>Pre-release Trigger to CPSM</td>
</tr>
<tr>
<td>TRG2</td>
<td>OPSM</td>
<td>O&lt;sub&gt;IR&lt;/sub&gt;</td>
<td>Trigger from Parent Object</td>
<td>O&lt;sub&gt;TR&lt;/sub&gt;</td>
<td>Post-release Trigger to CPSM</td>
</tr>
<tr>
<td>TRG3</td>
<td>CPSM</td>
<td>C&lt;sub&gt;tr&lt;/sub&gt;</td>
<td>vt from Agreement Protocol</td>
<td>C&lt;sub&gt;TR&lt;/sub&gt;</td>
<td>No Action</td>
</tr>
<tr>
<td>TRG4</td>
<td>CPSM</td>
<td>C&lt;sub&gt;IR&lt;/sub&gt;</td>
<td>vt from Agreement Protocol</td>
<td>C&lt;sub&gt;TR&lt;/sub&gt;</td>
<td>‘agreement’ to Parent Agent</td>
</tr>
<tr>
<td>RLS1</td>
<td>OPSM</td>
<td>O&lt;sub&gt;tr&lt;/sub&gt;</td>
<td>Release from Parent Object</td>
<td>O&lt;sub&gt;IR&lt;/sub&gt;</td>
<td>Release to CPSM</td>
</tr>
<tr>
<td>RLS2</td>
<td>OPSM</td>
<td>O&lt;sub&gt;TR&lt;/sub&gt;</td>
<td>Release from Parent Object</td>
<td>O&lt;sub&gt;TR&lt;/sub&gt;</td>
<td>Release to CPSM</td>
</tr>
<tr>
<td>RLS3</td>
<td>CPSM</td>
<td>C&lt;sub&gt;tr&lt;/sub&gt;</td>
<td>Ar from Agreement Protocol</td>
<td>C&lt;sub&gt;iR&lt;/sub&gt;</td>
<td>No Action</td>
</tr>
<tr>
<td>RLS4</td>
<td>CPSM</td>
<td>C&lt;sub&gt;TR&lt;/sub&gt;</td>
<td>Ar from Agreement Protocol</td>
<td>C&lt;sub&gt;TR&lt;/sub&gt;</td>
<td>‘agreement’ to Parent Agent</td>
</tr>
</tbody>
</table>

Table 3.2: Actions Caused by Control Signals in CPSM/OPSM Interaction.
Figure 3.6: The Logic of Agreement Protocol
protocol box will be initialized to zero. Thus, it follows that at the beginning of each activity cycle each of these registers will have a zero in it.

Any time a release signal is received its corresponding register will be set to 1. However, trigger signals are stored in registers only after certain preprocessing takes place as per the following logic: The $i$th trigger signal $t_i$, where

$$t_i = ((r_i \land t_{\text{post}}_i) \lor t_{\text{pre}}_i)$$

will set the $i$th trigger register. The logic shown above will guarantee that all late post-release signals, i.e., those belonging to the previous activity cycle, will be ignored. This is because, as shown above, a $t_{\text{post}}_i$ will be accepted only if its corresponding $r_i$ had been already set to 1. Since all release registers were reset to zero before the beginning of an activity cycle, $r_i$ will be equal to 1 if and only if a corresponding release signal had been received in the current activity cycle. Thus, the trigger register $t_i$ will get set to 1 if and only if either a pre-release trigger was received or a post-release trigger intended for the current activity cycle was received from one of the client Opsms.

The first trigger register, say $t_1$, that gets set to 1 will cause output $\forall t$, shown in Figure 3.6, to change value from 0 to 1. This change will be interpreted as a trigger input by the Cpsm sequential machine.\(^8\) Similarly, the output line $\forall r$ in Figure 3.6 will change its value from 0 to 1 only when all registers $r_i$ for $1 \leq i \leq n$ get set to 1. The agreement signal in the figure will change its value from 0 to 1 when both $\forall r$ and $\forall t$ are 1. The change in the value of the agreement signal will reset all the registers in the agreement protocol logic, as mentioned earlier. At about the same time the parent agent will also receive the ‘agreement’ signal shown in Figure 3.5. This will cause the parent agent of the Cpsm to end the current activity cycle.

\(^8\)Since the sequential machines are here assumed to be asynchronous they are sensitive only to changes in values of inputs and not to steady state values.
Reader may verify that the `agreement` output shown in Figure 3.5 is logically equivalent to the `clear signal` shown in Figure 3.6. However, these two signals may not occur at the same time. There is an implicit (and a reasonable) assumption here that clearing of registers in the agreement logic box will always occur before the parent agent of CPSM terminates the current activity cycle. However, as the reader may note, there is a potential race condition here between the `clear signal` in Figure 3.6 and the `agreement` signal in Figure 3.5. It is, in principle, possible that the `agreement` signal appeared before the `clear signal` and the next activity cycle began before all the registers in Figure 3.6 got cleared. This race condition may, of course, be eliminated by setting,

\[
AGREEMENT = (\text{`agreement'} \land \neg r \land \neg t),
\]

where `agreement' is the agreement signal in Figure 3.5 and \( \forall r = (r_1 \lor r_2 \lor \ldots \lor r_n) \), where \( r_i \)'s are the release registers in Figure 3.6, and similarly with \( \forall t \). This will guarantee that the \text{AGREEMENT} signal will reach the agent only after all registers in the agreement protocol logic box have been cleared.

The agreement protocol logic described here achieves several important objectives:

1. It guarantees that race condition problems are avoided.

2. It guarantees that every client of a composite agent would have completed processing input data and written its output before the active period of the agent's activity cycle is terminated.

3. By allowing for variable length registers in the agreement protocol box one may dynamically change the number of clients serviced by an agent, even in the middle of the active period of an activity cycle, as mentioned earlier.

Reader may verify that the composite agent port behavior will reduce to that of the single (agent-port, obj-port) case shown in Figure 3.2 when only one client is
Variations on the above protocol between (agent-port, obj-port) to accommodate other possible port pairings like (ring-port, ring-port), (agent-port, ring-port), (watch-port, obj-port), etc. interactions are presented, in the next subsection.

3.4 Other Port to Port RESTCLK Protocols

Ring-port, ring-port Protocol: As shown in Figure 3.7 this protocol is used for allowing a data bag to migrate around a clock ring or the delivery side of a watch ring. Port sequential machines in this case have only two states: Active and Inactive states, $S_A$ and $S_I$, respectively. When an agent is idle its ring-port will be in the $S_I$ state and the agent will not have communication control over its parent ring. When the agent is active, the ring-port will be in the $S_A$ and the agent will be in control of communication over its parent ring. Transition from $S_I$ to $S_A$ will occur when a ring port receives an advance signal. The transition from $S_A$ back to $S_I$ will occur when the ring-port sends out an advance signal. Communication will be, of course, always only one way.

Agent-port, Ring-port Protocol: This protocol is used for data migration
Figure 3.8: Illustrating Agent-port to Ring-port Communication.

from an agent-port to a ring-port on the delivery side of a watch ring (see the ring-port on the delivery side in Figures 2.4 and 3.8). In Figure 3.8 two observer agents are shown attached to the delivery side of a watch ring, just to show that it is possible to have two such agents intercepting data on the delivery side and passing it on to observers. This is not a requirement. In this protocol an agent-port (the port on top in Figure 3.8) will send an advance to the ring-port while it is moving to its active state $C_{tr}$ (see transition from $C_{TR}$ to $C_{tr}$ in Figure 3.5). After sending the advance the agent-port will continue to remain in its active state. The ring-port will never communicate back with the agent-port from which it received the advance. Thus, ring-ports on the delivery side of watch rings will behave exactly in the same manner as ring-ports on clock rings. Again, communication is only one way.

**Ring-port, Obj-port Protocol**: This is used again on the delivery side of a watch ring (see again Figures 2.4 and 3.8) between an agent on the watch
ring and the obj-port of the watch ring. Communication is in one direction only. Here when the ring-port of the agent (the second agent $A_k$ in Figure 3.8) sends an advance to obj-port (the port at the bottom of Figure 3.8) the ring-port will become inactive and the obj-port that receives the advance will become active. On receipt of advance the obj-port here will behave exactly in the same manner as in the (agent-port, obj-port) protocol discussed earlier. The obj-port will never communicate back to the ring-port; communication is one way.

**Obj-port, watch-port Protocol:** This is used on the dispatch side of a watch ring (see Figures 2.4 and 3.9). In Figure 3.9 two observer agents are shown attached to the dispatch side of the watch ring in order to indicate that it is possible to have two such agents intercepting data on the dispatch side and delivering it to observers. This is not a requirement. Communication is again in one direction only. When obj-port sends *pre-release trigger* signal (trigger before release) the watch-port will simply pass it along to the next entity on the watch
ring (which can be another watch-port or an agent-port), and transfer its own internal state to $W_{TR}$ state (i.e. trigger received but no release received).

When obj-port sends a release signal, the watch-port state will not change, but the parent watch-agent (agent $A_k$ in Figure 3.9) attached to the watch-port will respond to this release signal in the same way as a clock-agent will on receipt of an advance from its previous agent: Thus $A_k$ will forward data it received from the obj-port to all of its observers by sending an advance to all of them.

When a watch-port receives a post-release trigger (trigger after a release), if it is in the $W_{tR}$ state then it will simply pass it along to its next entity and will change its state to the $W_{TR}$ state. However, if it is in the $W_{tr}$ state, then the watch-port will make a transition to a special state, say $W_{Xr}$ (X for “transient”) in which it will remember that a post-release trigger has been received. Reader may note at the time this trigger is received, observers being serviced by agent $A_k$ will all already have output data sent by the obj-port.

When the parent watch-agent $A_k$ of the watch-port senses satisfaction of the agreement protocol among the observers it services, it will send a release to its watch-port. The watch-port at this time could be in one of the states $W_{tr}$, $W_{TR}$ or $W_{Xr}$. On receipt of release from its parent watch-agent $A_k$, if the watch port was in state $W_{tr}$ it will change its state to $W_{tR}$ and in this case pass along the release signal to its next entity on the dispatch side, which could be another watch-port or an agent-port. If it was in state $W_{TR}$ than it will change to state $W_{TR}$ and pass along the release in the same manner. If it was in state state $W_{Xr}$, then it will change to state $W_{TR}$ and pass along to the next port or the delivery side both signals release and the post-release trigger in that order.

Thus the composite agent-port on the watch ring to which the watch-port is attached will continue to behave as though it is directly communicating with the obj-port of the ring. The presence of watch-ports on the delivery side of a watch ring will not affect the logic of control signal flow on the watch ring, even though
it may affect their timings.

**Watch-port, Watch-port Protocol:** The possibility for this is illustrated in Figure 3.9. The protocol is the same as the (obj-port, watch-port) protocol presented above.

**Watch-port, Agent-port Protocol:** This is also illustrated in Figure 3.9. Here the watch-port of agent $A_j$ is communicating with the agent-port of agent $A_i$. The protocol is same as the (obj-port, agent-port) protocol. The last watch-port on the delivery side of a watch ring will always communicate with the agent-port of that ring.

**Agent-port, Rep-port Protocol:** This is used when an agent-port communicates with one of its representatives in another machine. This is illustrated in Figure 3.10. The protocol used here is identical to the (agent-port, obj-port) protocol discussed earlier and illustrated in Figures 3.2 and 3.3. As in a watch ring,
communication here is bidirectional. However, it will occur \textit{via} an inter-machine communication net.

This completes our discussion of all the RESTCLK protocols used for communication between various kinds of RESTCLK ports. In the next chapter we will discuss how RESTCLK may create and destroy its network components dynamically without any awareness or collaboration from application system objects. We refer to this as \textit{transparent dynamicity} of RESTCLK, because application objects cannot see the dynamic changes taking place in a communication network.
Chapter 4
Dynamicity and Transparency

We discuss here the basic concepts that make possible dynamic and transparent creation and destruction of RESTCLK network components. As mentioned earlier, the creation and destruction of RESTCLK components will always be carried out only by the RESTCLK system, when requested by application objects. To do this RESTCLK should, of course, have the capability to communicate with both application objects and RESTCLK network objects.

Changes in a network will occur when new network components are introduced or existing components are removed. To guarantee transparency, introduction and removal of components should not (a) violate application system consistency and (b) interfere in any manner with ongoing application system processes. In order to achieve this RESTCLK should be able to obtain the current states of ports which are involved in coordinating communications in regions of network in which changes are being made, at the time changes are being made. To obtain such state information RESTCLK will communicate with its agents and ports using communication facilities provided by the operating system in whose environment RESTCLK itself has been implemented. A central requirement on RESTCLK system is, delays involved in such communications should not cause RESTCLK to get a distorted view of the states of ports: thus at all times, information on port states received by RESTCLK should be identical to the true current states of the ports.

1The concept of application system consistency is described later in Section 4.6.
In the current implementation of RESTCLK this requirement is satisfied by making agents and obj-port information a part of the RESTCLK system itself. Information about obj-port, agent and agent-port states are all maintained internal to RESTCLK. Thus at this point the RESTCLK paradigm is applied only to application system communications, not to RESTCLK communications. All communication between RESTCLK and its agents, various ports, representatives and objects will use host operating system facilities. Thus, these communications will not be available for observation and control. We will later discuss in Section 8.3 extensions in which RESTCLK and operating systems, and even the hardware of a computing system, may all be themselves implemented within the RESTCLK paradigm.

4.1 Installation of Objects

RESTCLK comes with a special seed Object which may be used to start up application systems. Initially this seed object may be setup to create and install application objects needed for system start up. Once this is done, any object may request RESTCLK to create and install objects. We describe below informally the protocols that coordinate object creation and installation.

Request for Object Creation: As mentioned above, this can come from any application object. The request will contain the following information: (i) object name, (ii) identities and handles of I/O ports to be created for it (iii) location information which will consist of Machine ID & RESTCLK manager ID in the machine where it is to be installed, (iv) handle containing security classification of the new object, if any, which a requester might have provided and (v) the requesters ID and handle. RESTCLK will use this information to create a new

\footnote{It should be noted that in a distributed system each machine may contain one or more instances of RESTCLK manager. Each instance of RESTCLK manager will have control over all the application and RESTCLK network objects which are in its domain.}
RESTCLK ID for the object and associate with this ID a handle. This handle will contain the object ID and its intended security classification, if any.\(^3\)

**Creation & Installation:** RESTCLK will pass the object handle together with the handle (or ID) of the requester to the application system’s security manager. It will get back from the security manager an approval or disapproval for object creation. If it is approved it may also obtain a security classification for the object, if one had not been already specified, together with classifications for all the I/O ports to be installed on the object. Classifications for the I/O ports need not all be the same, but they should all have *security compatibility* with the classification assigned to the object.\(^4\) This is a responsibility of the application system security manager. Once permission for creation is obtained, RESTCLK will request the operating system to create and install the object.

When the object is created its generic initialization routine will begin execution. This initialization routine will automatically create and install the designated ports for the newly created object and all the requested I/O ports. All these ports will now have unique handles associated with them, each with an ID and a security classification and RESTCLK will already have all of this information. The port sequential machines of all the ports of the new object will be initialized to the idle state \(O_{IR}\). Of course, this state information will be simultaneously available to both the parent object of the ports and RESTCLK.

At this point all newly created entities will have unique identities provided by RESTCLK, and RESTCLK itself will have access to the handles associated with these identities. This will complete the object creation process.\(^5\)

---

\(^3\)In the following we will assume that the handle of an entity will always contain the entity ID and its security classification.

\(^4\)The concept of *security compatibility* between security classifications is discussed in Chapter 7. Roughly speaking, a pair of entities, \((X, Y)\), are said to have *security compatibility* only if \(X\) can communicate with \(Y\).

\(^5\)In the above discussion we have assumed operating system will provide the basic facilities needed for creating and installing objects and object initialization routine will provide facilities
4.2 Creation and Destruction of Obj-ports

Once an object has been created and installed, during its life time the object may create additional ports for itself or it may destroy any of its existing ports. Every time an object thus modifies its port structure, it should obtain permission from RESTCLK. At the time RESTCLK grants permission to create an obj-port it will specify the port’s handle. This will enable RESTCLK to setup facilities for checking security features associated with obj-ports whenever need arises. Again, all such newly created obj-ports will be initialized to state $O_{tr}$. An object may destroy one of its existing ports only if that port is not attached to a watch ring, i.e., the port is totally disconnected from the RESTCLK network.

4.3 Installation of Clock Rings

Any object may request creation of clock rings. Clock rings do not have a security classification associated with them.\(^6\) If creation of the new clock ring requested by the requesting object is not disallowed by the security layer of RESTCLK, then RESTCLK will create the new clock ring. At the time the clock ring is created RESTCLK will also create an agent and associate a handle with the agent. We will refer to this agent as the primary agent of the clock ring.

Any object having appropriate privileges, may request a clock ring to be destroyed. It can be destroyed only when the only agent attached to the clock ring is its primary agent and the primary agent is not attached to any watch ring. The primary agent of the clock ring will always get destroyed when the ring itself is destroyed.

It should be noted all agents will always have handles associated with them.

---

\(^5\)Clock rings in RESTCLK are just passive data structures; not active entities. Generally, the classification associated with the primary agent of a clock ring will also be associated with the clock ring itself.
4.4 Installation of Agents

Any object may request RESTCLK to create a new agent on a clock or a watch ring, or request destruction of existing agents on the ring. An agent may be destroyed only if it is not connected to any watch ring and it is not the primary agent of its parent ring.

A new agent can be created only if its attachment location on a ring has been specified. This location will be specified by declaring the ID of agent who would be just before the new agent’s intended location on the ring. Let $A_i$, be the ID of the reference agent that specifies the intended location of the agent to be newly created. RESTCLK will seek permission to create the new agent from security manager by providing to the manager the following information: The new agent’s handle, say $H$, (which may or may not contain its security classification), its location $A_i$, and the handle of the object that requested its creation. The security manager will approve the request only if the requesting object has the authority to request creation of such an agent, and the classification in the agent’s handle, if already specified, has \textit{security compatibility} with those of its location $A_i$ and its next agent $A_{i+1}$. If no classification for the new agent had been specified then the security manager will supply a classification that is compatible with those of $A_i$ and $A_{i+1}$.

If creation and attachment of the new agent is not disallowed then RESTCLK will create and install the agent at its specified location. At the time this newly created agent is attached to its parent ring it will not have communication control over the ring.

4.5 Installation of Watch Rings

A watch ring will connect an obj-port to a composite agent-port. The creation and installation of a watch-ring will be governed by the states of port sequential
machines, OPSM and CPSM. We will test for state compatibility between CPSM and OPSM states before tuning them to each other. The state compatibility relation used for this purpose is defined in the next subsection. We will hereafter refer to state compatibility simply as compatibility in the rest of this chapter.

4.5.1 Compatibility of Agent-Port, Obj-Port States

*Syntactic Definition of Compatibility:* A pair of states \((C_{\alpha \beta}, O_{\delta \gamma})\) for \(\alpha, \delta \in \{t, T\}\) and \(\beta, \gamma \in \{r, R\}\) are compatible if and only if they can occur together during a communication session without violating the RESTCLK protocol described in Section 3.3.

*Semantic Definition of Compatibility:* A pair of states \((C_{\alpha \beta}, O_{\delta \gamma})\) for \(\alpha, \delta \in \{t, T\}\) and \(\beta, \gamma \in \{r, R\}\) is compatible if and only if each port in its current state is expecting to receive signals that other might send, as specified in the RESTCLK protocol.

Compatibility will thus guarantee that when a newly created watch ring is connected between an obj-port and a composite agent port in order to tune them to each other, (a) the inclusion relationship, mentioned earlier, will not be violated and (b) communication control over the watch and clock rings will reside with the correct ports, as required by the RESTCLK protocol. As discussed in Section 4.6, it is the preservation of these properties (a) and (b) that lead to transparency.

The compatibility relation is shown in Table 4.1. The table shows the pair of states, \((C_{TR}, O_{TR})\), as being compatible. This because both are in idle states: Certainly RESTCLK protocol allows such a pair of states to co-exist. Agent-port in state \(C_{TR}\) is not expecting any signals from obj-port that will require it to take actions. Obj-port is also not going to send any signals. Establishing a connection between them will not change their mutual expectations and will not interfere with the internal state of the parent object of the obj-port.
The table shows states \((C_{TR}, O_{tR})\) also as being compatible with each other. Again, as the reader may verify, such a pair may co-exist as per RESTCLK protocol in group communication. In state \(C_{Tr}\), the composite agent port has received at least one trigger from at least one of its existing clients but has not received releases from all of its clients. Therefore, the composite agent port is expecting to receive both releases and triggers from its clients. The obj-port in state \(O_{tR}\) may send a trigger. It has already sent a release. Thus, in terms of signals that the two ports may exchange the expectations of the two ports do not conflict with each other. This guarantees that when watch ring connection is established it will not interfere with the internal state of the parent object of the obj-port. This is the reason they are compatible.

Reader may similarly verify validity of the other compatible pairs shown in the table. Let us now consider an example of incompatible pair of states.

States \((C_{TR}, O_{tR})\) are clearly incompatible. They certainly cannot co-exist as per RESTCLK protocol, because in state \(C_{TR}\) the composite agent port will not expect to receive a release from any of its clients. All the parent objects of obj-ports tuned to the composite port have already completed their respective tasks and the active period of their most recent communication session has ended. However the obj-port that is to be newly tuned to the composite port is in state \(O_{tR}\). Thus, once connected it may send a release contrary to the agent-port’s expectations. In this case, the agent will not be able to take appropriate actions.
in response to this signal, namely the agent will not be able to read and bag output data that the parent object of the new obj-port might have written into the private memory of the new obj-port. Thus, this data will be lost. This may, of course, interfere with the internal operations of both the parent object and the intended recipient of that data. Therefore, these two states are not compatible. Reader may similarly verify that the other pairs of states shown in Table 4.1 are also indeed incompatible with each other.

4.5.2 The Tuning Process

Always a watch ring attachment will cause an obj-port to be tuned to an agent-port: As a result of this tuning process the obj-port will have access to the private memory of the agent and agent will have access to the private memory of the obj-port during active intervals of all activity cycles. This will be a direct consequence of the security classifications association with these two entities. Thus always a watch-ring may be installed only if the ports connected by the watch-ring have mutual security compatibility.

When introducing or removing a watch ring care should be taken to prevent interference with ongoing processes in the affected objects. RESTCLK will do this by making sure the state of the obj-port and the composite agent-port to be connected by a newly created watch ring are either compatible with each other before the connection is made, or can be made compatible immediately after making the connection within one atomic operation. Thus making a connection may require RESTCLK to take some actions. These are described in this subsection.

When the states are incompatible there are two cases to consider:

- **InComp** (i): By taking certain actions immediately upon making the watch ring connection the incompatibility can be resolved.

- **InComp** (ii): There is no way to resolve the incompatibility.
Similarly when the states are compatible, if the composite agent-port is currently holding communication control over its parent ring (i.e., it is currently in the middle of an ongoing communication session) and the obj-port to be newly tuned is in one of its idle states, then also there are two cases to be considered, depending on application system requirements:

- **Comp (i):** The obj-port may be made active immediately in order to participate in the current ongoing communication session of the agent, or

- **Comp (ii):** The obj-port may be allowed to become active only in the next communication session of the agent. This option is shown in Table 4.2 entries as an alternative in the form “or C,” where appropriate.

Table 4.2 specifies actions taken by RESTCLK for the various combinations of states of the two ports which are to be connected by a new watch ring. Actions are denoted by notations in the table, where C stands for Connect, N stands for New composite port state, A stands for sending an advance to obj-port and W stands for Waiting until obj-port and agent-port move to a compatible pair of states. Notation combinations such as, say NCA, will indicate the following actions: going to a New composite port state, sending an Advance and Connecting. Similarly with other notation combinations. For combinations NC and NCA, the \( \langle \text{ActionCombination}, \text{NewState} \rangle \) pairs shown in the table indicate the new state to which the composite agent port will go for the indicated ActionCombination.
The various actions are summarized below.

**C** In this case the newly created watch ring may be immediately connected to the two ports. No additional actions are necessary.

**CA** In this case immediately after making connection RESTCLK will cause the composite agent port to send an *advance* to the newly connected obj-port. Sending this advance will put the new obj-port into state $O_{tr}$, which will make it compatible with the agent-port state in all cases shown in the table above.

The only case where this action is required is for the pair of incompatible states, $(C_{tr}, O_{TR})$. In all other cases sending an *advance* is optional as stated in case ‘Comp (ii)’ above.

**NC** Here the composite agent-port is set to a new state by RESTCLK just before making the connection in order to make the state of the agent compatible with that of the obj-port. The table indicates in each case the new state to which the composite agent port will go.

**NCA** In this case also, as in case **CA** above, RESTCLK will cause the composite agent port to send an *advance* after making the watch-port connection. Also, as in case **NC** the composite agent port will move to the new state indicated in the table just before making the connection. This new state will be compatible with the state to which obj-port will move, when it receives the *advance*.

In the case of state pair $(C_{tr}, O_{TR})$, as per case ‘Comp (ii)’ mentioned earlier, instead of taking action **NCA** RESTCLK could have simply taken the action **C**.

**W** This is the incompatible case in which there are no actions RESTCLK could take to resolve the incompatibility. In this RESTCLK may either refuse the
request and expect the requester to resubmit the request at a later time, or RESTCLK may periodically poll the port states until compatibility could be satisfied. The latter option may also use a timeout.

Two cases where W action occurs appear in \((C_{TR}, O_{Tr})\) and \((C_{TR}, O_{tr})\) in Table 4.2. In both these cases new obj-port is active and the composite agent port is idle. In this case RESTCLK will wait until either obj-port becomes idle or the agent-port becomes active or else return a failure to the requester.

Before making a watch ring connection RESTCLK will check the obj-port and composite agent-port states for compatibility. If action W is called for then, of course, nothing would be done. Otherwise, first action done will be the action N to change the state of the composite agent port to its new state, if such action is called for as per Table 4.2. Immediately after taking action N action C will be carried out. Of course, it is quite possible that by the time action C is completed the port states could have changed. But as shown in Figure 4.1 none of the possible state changes that may occur while action C is being performed could ever bring the port states back to incompatibility. This is explained below.

In Figure 4.1 each node is an \((agent\text{-}port, obj\text{-}port)\) state pair. The solid arrows show the possible state changes that can occur as a result of the RESTCLK protocol described in Chapter 3. The inputs that cause these state changes are shown as weights on these arrows. The inputs generated by the obj-port whose state appears in the nodes appear as ‘r’ (release) and ‘t’ (trigger). The inputs generated by another obj-port, other than the one whose state is shown in the nodes, appear in parenthesis. Thus, the state transition from state pair \((C_{tr}, O_{tr})\) to state pair \((C_{Tr}, O_{tR})\) will occur when the obj-port, whose state is shown in the node, sends out a release and another obj-port sends a trigger. Thus the arrow representing this transition in the figure has weight ‘r,(t)’. Similarly, the state change from \((C_{tr}, O_{tr})\) to \((C_{Tr}, O_{tr})\) will occur when an obj-port other than the
Figure 4.1: State Transition Diagram of Illustrating Compatible and Incompatible state pairs.
one whose state is shown in the node sends a trigger. Similarly, with all the other solid arrow transitions shown in the figure. These are all the transitions that may occur from one compatible pair of states to another.

The black nodes in the figure represent the incompatible state pairs for which \( W \) action is required as per Table 4.2. The five grey colored nodes are the incompatible state pairs for which it is possible to take some actions in order to establish the watch ring connection. The dotted arrows show the state changes on the composite agent-port that are forced by RESTCLK. Reader may note that no state changes are forced by RESTCLK on the obj-port and that in Figure 4.1 there are no transitions that occur from a compatible state pair to an incompatible pair. Thus, once RESTCLK brings an incompatible pair to a compatible one through action \( N \), the compatible pair can never revert back to an incompatible one. Thus during the time taken to perform the action \( C \) the pair of ports that are being connected by the new watch ring, once brought to a compatible state, cannot slip back to incompatibility.

Reader may verify, in both installation and destruction of watch rings, always it is only the agent-port that takes actions, when necessary, in order to maintain state compatibility. No collaboration is expected or required from the obj-port or its parent object.

4.6 Transparency

The only situation where a dynamic network reconfiguration has an opportunity to interfere with ongoing operations of objects in an application system arises in the context of establishing or removing watch ring connections between existing agent-ports and obj-ports. Because, it is this process that connects (disconnects) an obj-port to (from) the RESTCLK network. By preserving state compatibility
during such changes RESTCLK makes sure that such changes will not affect synchronization and coordination among communicating ports in the system. Also, by vesting all responsibility for state compatibility preservation on RESTCLK agents only, RESTCLK makes it unnecessary for application objects to either know about such changes or collaborate with RESTCLK in order to implement such changes. We refer to this as the transparency property. One may claim, as a result of this transparency property, none of the dynamic network reconfigurations that change a RESTCLK network will affect synchronization and coordination among communicating objects in an application system.

When watch ring attachment (or removal) process is executed as per specifications given above, the only effect that a parent object might see is a delay in receiving input data; data can be received only after connection has been established. It should be noted that the parent object could always write output data into the obj-port's memory, whether the obj-port is connected to a watch ring or not. Thus sending of outputs by a parent object is never affected by watch ring attachment and removal processes because state compatibility is always preserved. However, the receipt of this output by its intended recipient might be delayed in the case of watch ring attachment (or prevented in the case of removal) since data would be received only after watch ring connection had been established.

When a watch ring is removed one of two possible cases may occur: Either the parent object of the disconnected obj-port does not receive input data, or the output written by the parent object via the disconnected obj-port is not forwarded to its intended recipient. In either case, it would be the intended effect and thus one may assume that disruption would not occur.

As discussed in the previous subsection, by ensuring compatibility of port states that are being tuned to each other, after establishment of watch ring connection RESTCLK guarantees that ongoing application object activities will

\footnote{It may be noted that before destroying a watch ring connection the states will always be}
never be affected, but for possible input/output delays.

As a consequence of the transparency property, in RESTCLK there is no need to collaborate with application objects while making changes in the RESTCLK network. Also no special actions are necessary to preserve application system consistency. This refers to compatibility of the internal states of application system objects that might be involved in exchanging messages. A system is said to be in a consistent state if all objects in the system have the correct expectations relative to sending and receiving of messages among themselves. RESTCLK would preserve these expectations, in spite of dynamic network structure changes, as discussed earlier in this chapter.

This is the principal benefit of dynamicity with transparency. It not only can simplify object designs but can also allow considerable design flexibility. Some of the significant benefits accrued from this flexibility are discussed in Chapters 5 and 6.

The significance of RESTCLK to distributed systems design and implementation is briefly discussed in the next chapter.
Chapter 5
Significance to Distributed Systems Development

5.1 Input/output Isolation

One may think of obj-ports as providing high impedance interfaces between objects and their environments, in the sense that message traffic through obj-ports would not interfere with scheduling of internal activities of the object. When a new message arrives the parent object need not interrupt its activities in order to attend to the incoming message. When an expected message does not arrive at the time an object polls its obj-port then the object can just skip that obj-port and go on to service the one next to it. In all cases an object may attend to its inputs driven only by its own internal schedules, i.e., when it has completed doing, whatever it had been doing. After processing inputs at its own time and sending its outputs through an obj-port an object may then trigger or release the obj-port. The obj-port will subsequently forward the outputs to the agent-port tuned to it. Each agent-port will wait for this to happen before taking any other action. This in effect isolates input/output activities from the internal activities of the object allowing internal activities and I/O activities to occur concurrently without mutual interference.

In the examples discussed in Chapter 9 we present some pseudo-codes in which we show application objects telling their obj-ports to send or receive control signals. This kind of explicit encoding of RESTCLK control signals inside application objects is not a requirement since obj-ports may be used to provide
interfaces between RESTCLK networks and application objects which were not written for RESTCLK environments.

Thus an obj-port may interpret an application object’s *Send*, *Write* or *Output* commands as signals to send output data to its private designated memory, and emit *trigger* and *release* signals as appropriate. Similarly, it may interpret an application object’s *Receive*, *Read* or *Input* commands as signals to look for *advance* at the obj-port and copy data from its agent’s private designated memory.¹

### 5.2 Connection Oriented Architecture

In a RESTCLK network an application object may communicate with another if and only if a communication pathway is established beforehand between obj-ports belonging to the objects. Such a connection oriented architecture is also used in the POLYLITH software bus [38]. This is to be contrasted with *connectionless* communication architectures as in RPC [37, 4, 46]. In RPC establishment of pathways between senders and receivers is not a responsibility of an application system.

We have assumed in RESTCLK that *management objects* of an application system will be responsible to make use of RESTCLK’s services to establish and remove communication pathways dynamically, as and when needed.² Management objects in an application system would thus be responsible for initial configuration and any dynamic reconfigurations of the application system. An important feature in RESTCLK is that at any time existing management objects may be

---

¹A more general scheme in which interfaces between obj-ports and their external communication environments are specified in an *Interface Description Language* (IDL) and these specifications are compiled into *Port Drivers* inside obj-ports, is described in OMSOFT [43], which provides a general distributed systems development environment using RESTCLK communication networks.

²Viewing an application system in terms of application objects and management objects is not new. This widely accepted method of organizing application systems has been proposed in the CORBA standards [16].
changed, or totally new ones introduced, without having to change application objects in any manner. A total decoupling between management and application objects is thus achieved in RESTCLK network environments.

Although the operations for creation and destruction of communication pathways will be invoked by the management objects in an application system, these operations will actually be carried out by the RESTCLK system itself. This makes it possible for RESTCLK to control and establish unique identities for the network components it creates. This in turn enables an application programmer to extend the security environment of an application system to RESTCLK communication network as discussed in detail in Chapter 7 and briefly outlined below.

### 5.3 Security Environment of RESTCLK Network

One possible method an application system programmer may employ to extend an application system security environment to RESTCLK network is to assign to each network component a security classification, at the time the component is created. We assumed this to be the case in our discussions in Chapter 4. RESTCLK will submit to security enforcement facilities provided by an application system, information on security classification in order to determine when and where network pathways may be created or destroyed. Thus, management objects in an application system will get approval from the application system's security enforcer before changes are made to a RESTCLK network. We give below an example of the kind of policies that might be used to thus extend the security environment of an application system to the RESTCLK network.

An obj-port might inherit the security classification of its parent object. A watch-ring might inherit the classification of the obj-port connected to it. A

---

3Since the parent object here is a part of an application system, we assume that it will always have a security classification.
composite agent-port might inherit the classification of the most secure obj-port tuned to it. A clock-ring might inherit the classification of the most secure agent attached to it. Once such a classification of network objects is established, then one might allow new agents to attach themselves to existing rings only if the classification level of the new agent is higher or the same as that of the ring itself, unless special permission is granted by the security enforcer of the application system.

We do not claim here that this is the only possible policy that might be employed. Security classification of application system objects may be extended to RESTCLK networks in a variety of ways. Also, it is possible to change classification levels associated with RESTCLK components dynamically as and when needed, based upon application system requirements. Setting up laws of application system security and protection, and systems for their enforcement, would be the responsibility of application system programmers. RESTCLK simply provides the facilities for extending these application system features to RESTCLK communication networks. RESTCLK by itself does not impose restrictions on the manner in which such security environments may evolve. A general scheme for extending security environments of application systems to RESTCLK networks is discussed in Chapter 7.

5.4 Communication Transparency

In an application system using RESTCLK only management objects might know who is communicating with whom. Application objects themselves will not know identities of other objects with whom they might be communicating. We refer to this as communication transparency.

This makes it possible to program application objects in a manner that is independent of network contexts in which they might be used. In the terminology
introduced earlier programming in the small thus becomes independent of configuration programming. This is largely responsible for making I/O-functionalities of an application object independent of its C-functionalities in the context of RESTCLK communication network.

This feature together with the connection oriented architecture, mentioned above, makes it possible to dynamically reconfigure application systems even if such reconfigurations have not been a priori planned for. This capability is important also for fault management and emergency handling: For fault management in the RESTCLK context one needs only facilities (monitors) for detecting errors and failures. When unanticipated errors or failures are detected, one may dynamically determine compensating strategies to restore correct functioning and reconfigure the application system appropriately. The same holds also when need arises to react to unanticipated emergencies. These are discussed in greater detail in Chapter 6.

5.5 Active RESTCLK Components

Ports in RESTCLK are active in the following sense: Port sequential machines described in Chapter 3 enable ports to participate in coordination and synchronization of communication. Normally RESTCLK expects its agents to simply gather data into a bag and forward the bag along existing pathways. It is possible to extend this behavior to allow agents to optionally apply any application specified function to the data. Such application dependent functions may impose special security encoding on data at specified times on selected pathways, may selectively suppress forwarding of data to certain clients, may modify data in ways that are appropriate for delivery to designated clients, etc. These facilities may allow for a variety of dynamically determined security policies to be superimposed on data migrating over a RESTCLK network. This feature is discussed in greater
detail in Chapter 7.

5.6 Unit Buffering and Lossless Communication

RESTCLK uses reply semantics in the following sense: Once data is sent out by an obj-port in one activity cycle, it can send out a next set of data only in its next activity cycle. The *advance* signal that marks the beginning of the next activity cycle may here be construed as a reply to the data sent in the previous one.

Data in this unit buffer\(^4\) may, of course, get modified as it migrates around the ring. RESTCLK protocol ensures that for each client object that is entitled to receive data from this buffer, the buffer would remain unchanged until that client object had explicitly acknowledged the receipt and use of the data, through a *release* signal. In this sense communication in RESTCLK is lossless and buffer overflow will never occur. As we shall later see in Section 8.3.1, even with *pipelining*, which may require multiple buffers for each clock ring and allow more than one agent on a ring to be simultaneously active, communication in RESTCLK will still be lossless.

5.7 Group to Group Communication

RESTCLK is probably the only communication paradigm that allows communication between groups, where memberships in communicating groups may themselves be dynamically changed during any communication session. Group to group communication in RESTCLK is characterized by what we call *simultaneous dispatch and delivery*, as described below.

Data gathered from a sending group will be dispatched only when all members of the sending group have completed outputting their data. After gathering all

\(^{4}\text{This unit buffer in RESTCLK will be the memory } M_A \text{ associated with a clock ring and used by agents around the ring to communicate with their respective clients.}\)
data from the sending group, it is even possible to synchronize the sending of data with externally mediated trigger signals. Thus it is possible to dynamically set up a situation in which data gathered is forwarded, under certain circumstances, only on receipt of a trigger from a designated manager (temporarily) appointed dynamically for this purpose. We refer to this as simultaneous triggered dispatch.

Data received from multiple senders will always be delivered simultaneously to each member of the receiving group in a bag; we refer to this as simultaneous delivery. As mentioned earlier, this unique feature of RESTCLK has been exploited in OMSOFT [43] to implement a tetherless communication facility for negotiation among programmers in order to engineer complex software systems.

5.8 Observation and Control

Clearly, the ability to introduce agents on any RESTCLK ring at any time makes it possible to dynamically access data in transit, modify them and/or re-route them. This ability is at the heart of Observation & Control (O&C) in RESTCLK, which is discussed in greater detail in Chapter 6. We show there how O&C may be used in an wide variety of applications, such as dynamic reconfiguration, in situ testing, fault and emergency management, dynamic system evolution, etc. Examples that were implemented to demonstrate applications of RESTCLK O&C are discussed in Chapter 9. These examples were implemented and demonstrated in different versions of RESTCLK prototype as mentioned in Chapter 8.

5.9 Limitations

The protocol we have described above limits the number of active agents on a ring to exactly one. We will refer to the point on the ring at which an active agent is located as the activity point of the ring. Currently we allow each ring to have only one activity point at a time. This may force the obj-ports serviced
by agents on the ring to waste time, waiting for the next activity cycle to begin, leading to computational and communication inefficiencies. We will discuss in Section 8.3.1 pipelining methods which lead to increased efficiencies, by allowing multiple activity points to simultaneously exist on a ring.

Each communication pathway in RESTCLK, between any pair of obj-ports, requires two agents to mediate communication. This, of course, has the potential to cause more communication delays in RESTCLK networks than in other networks in which either only one agent is used or no agent is used to mediate communication. We would like to point out that any time one seeks to introduce greater control and versatility there is usually a price to be paid for that. What is important is to assess whether this price is worth paying. In view of the advantages mentioned above, we believe, it is worth paying. Later, in Chapter 8 we discuss and compare communication costs in various schemes and argue why the cost in RESTCLK is reasonable, when compared with the flexibility and dynamicity one gains.

In the next chapter we show how O&C may be implemented using RESTCLK facilities and discuss properties of RESTCLK O&C that make it useful in practical systems.
Chapter 6
Observation and Control in RESTCLK

6.1 Observation in RESTCLK

We outline in this section the different modes of data observation possible in RESTCLK and discuss how they may be used to dynamically control configuration and computation in application systems. Figure 6.1 illustrates the different modes of observation in RESTCLK: Parts A and C of the figure show the manner of attachment of observation networks for observing data flowing, respectively, out of and into a communication group. Parts B and D show the manner of attachment of observation network for observing data flowing, respectively, out of and into a particular obj-port. As shown in part A, the observed group may be attached to either a clock ring or a watch ring. Groups attached to agents on clock rings will usually be application system objects. Groups attached to agents on watch rings may be observers themselves. Thus the figures show how observation in RESTCLK may be used recursively on observation networks themselves.

In cases B and D, where data flow through a particular obj-port is being observed, the observing agents are attached to watch rings connected to the observed obj-port. In this case, as discussed earlier in Chapter 2, the observing agent will be attached to the delivery side of the watch ring through a ring-port and attached to the dispatch side through a watch-port.

The observation networks in these diagrams show only one observer object in each. This is not a requirement in RESTCLK. The same observed data can be shared by a group of observer objects.
Figure 6.1: Illustrating Different modes of Observation in RESTCLK.
The most important point to be noted in these diagrams is, in all of the cases establishment of the observation networks will not interfere with the activities of the observed entities. In fact, the observed entities may not even know that their inputs and/or outputs are being observed. This is a consequence of the *dynamicity and transparency* property of RESTCLK discussed in Chapter 4.

In general one would like any observation technique to satisfy the following desirable properties:

1. **Transparency**: Transparency will avoid having to design objects anticipating all future observation needs. Since observer networks can be dynamically installed without any participation from objects, RESTCLK satisfies this property.

2. **High resolution**: High resolution will guarantee that a system can be observed at any point to any desired degree of detail. Since our systems are assumed to be put together with objects, we will not expect to look inside the objects themselves. However, we will seek the ability to observe flow of data at any location in a communication net at any point of time. This property also is satisfied by RESTCLK.

3. **Recursion**: Recursion will guarantee that observers themselves can be recursively observed; thus no part of a communication net will be "blacked out" because of the introduction of observers. RESTCLK satisfies this property also.

4. **Minimal interference**: Minimal interference will guarantee that a system or sub-system, certified by testing in an observation environment, will not behave differently when it operates in its own intended operational environment.

This is a difficult property to satisfy. RESTCLK comes close to satisfying this property. We will be referring to this again in Subsection 6.3.1.
GOAL: Observe

Time needed for data to travel over the observation network

A

Time at which installation of observation network was completed.

Time at which data to be observed was received.

B

time

IMMEDIATE if and only if data delivery is not dependent on any other communication event.

Figure 6.2: Illustrating the Concept of Immediacy.

5. **Immediacy**: Figure 6.2 illustrates this concept. Time point A in this figure marks the time at which installation of observation network was completed and time point B marks the time at which the observer in the network receives the data to be observed. The interval (A, B) between the time points A and B is the time needed for the observed data to travel through its pathway in order to reach the observer.

Observation is said to **immediate** if and only if there is no need for any other communication event to occur during the interval (A, B), in order for the observer to receive the observed data.

It may be noted that none of the observation methods described in Figure 6.1 satisfy the immediacy property. Thus, these methods can be used only when immediacy is not a requirement for observation. We will later present a method
of observation through *probing* which satisfies the immediacy property.

Observation of data in this manner becomes significant to systems design and use only if it can be used in some manner to dynamically control graceful evolution of computations in application systems. To facilitate such control RESTCLK assumes that objects in application system will have certain specialized features that will make it possible for RESTCLK to observe characteristics which are internal to the objects themselves. The *designated ports* of objects introduced in Chapter 1 were intended for this purpose. The observation capabilities that these specialized features may provide to RESTCLK and their consequences in the RESTCLK context are briefly outlined below.

We have assumed that management objects may poll (and when necessary request objects to change) the internal states of application objects by using the designated *state ports* of the objects or by feeding the objects appropriate inputs. A management object may use *instrumentation* ports to query objects about measurements on computations, for which facilities have already been built into objects. *Emergency* ports are intended to be used when objects fail to execute properly. The manner in which emergency and instrumentation ports might be used will be dependent on facilities planned for *a priori* and built into an application system design.\(^1\) RESTCLK merely provides facilities to dynamically observe and use facilities of these kinds which have already been built into objects.

As always the design and implementation of management objects (or *managers*) in an application system will depend heavily on network contexts in which objects managed by the managers reside, and internal characteristics of such objects. Thus knowledge of object specifications, instrumentations and emergency handling facilities built into them, and knowledge of their network environments will all be needed to design and implement such managers.

\(^1\) The requirement to provide instrumentation and emergency handling facilities in objects is a well established industry standard and generally used in software industry [16].
In a RESTCLK environment, such managers may be deployed dynamically on demand without interfering with ongoing computations of an application system. They need not be \textit{a priori} designed for and installed in an application system. The significant new feature that RESTCLK makes possible is, design and deployment of these management objects need not be fixed \textit{a priori}. They can be designed, implemented and deployed as and when needs arise, at any time during the lifetime of an application system. By thus isolating and separating out the design and deployment of managers from design and deployment of the objects themselves, RESTCLK contributes to simplification of objects design, their portability and reusability. Thus, this is a significant contribution.

The different ways in which observation may be used in RESTCLK to control evolution of computations in application system are discussed briefly in the next section.

\section{6.2 Control in RESTCLK}

We shall consider here two kinds of control: Control to halt and restart computations in selected parts of a distributed system while maintaining consistency of the application system, and control to investigate and change states of computations as needed. These facilities may be used for testing and validation of system before it is released, for error recovery during operation, for performance tuning and monitoring, for dynamic reconfiguration, \textit{in situ} testing of new versions, etc. We will discuss some of these in the next section. Some of the desirable properties of techniques employed to exercise such control are presented in this section.

In general one would like any control technique to satisfy the following desirable properties:

1. \textbf{Minimal Disruption:} Since it is not always possible to exercise control with absolutely no disruption to the services provided by an application
system, it is desirable to find a way of exercising control in which only unavoidable disruption would occur. Later in this chapter we will explain how this may be realized in the context of RESTCLK.

2. **Minimal Planning**: It is in general difficult to exercise control without some prior planning. Thus, we require carefully thought out and planned design and construction of objects with designated ports. What is implied here is the following: Having designed objects with some facilities for monitoring and some for accepting externally induced changes, one would like a capability to use the designed facilities in all possible ways without having to plan for them *a priori*. Thus object designs need not anticipate all possible ways in which the facilities provided by objects might be used. RESTCLK allows minimal planning in this sense.

3. **Dynamic Instantiation**: It should be possible to dynamically instantiate sub-systems needed for control without having to halt system operations. RESTCLK has this capability.

4. **Selectivity**: It should be possible to exercise control selectively on any chosen subsets of objects in a system, such that objects external to the chosen subsets are not disrupted by the exercise of that control. What this property implies here is that it should be possible to exercise control selectively on any subset of objects in an application system for which such control is exercisable. RESTCLK by itself does not impose any restrictions on when and where such control may be exercised.

We discuss in the next section some applications which show how RESTCLK satisfies the above described desirable features of observation and control in a variety of significant cases.
6.3 Applications of RESTCLK O&C

6.3.1 Halting and Restarting computations

Assuming that an object can perform a computation only if all inputs necessary for the computation to proceed are available, to halt a computation driven by data received by an obj-port all that should happen in RESTCLK is that data being delivered to that obj-port should be blocked. This can be done in RESTCLK in two ways:

**Port Blocking with an Observer Agent**: Suppose a manager \( M \) wanted to halt computation in an object \( X \) that is driven by data received through one of its obj-ports \( p \). Let \( A \) be the agent delivering data to this port. Then as shown in diagram D of Figure 6.1, \( M \) will create an agent \( A_1 \) on the delivery side of the watch ring connecting \( A \) and \( p \), and tune one of its own obj-ports to this agent. Agent \( A_1 \) will now intercept data that is being delivered to port \( p \) of object \( X \) by agent \( A \) and deliver this intercepted data to \( M \). Once \( M \) gets hold of this data, it may hold on to this data by not sending a *release* to the agent \( A_1 \). This would prevent agent \( A_1 \) from forwarding any data to port \( p \). Thus \( M \) can halt computations in \( X \) driven by this data for as long as it wants.

**Port Blocking through Probes**: Another way of blocking is by preventing data dispatch from an obj-port. This is illustrated in Figure 6.3. In this figure obj-port \( p_1 \) will send data to obj-port \( p_2 \) *via* agents \( a_1 \) and \( a_2 \) through their parent clock ring. In the scheme communication control over the clock ring will periodically switch between the two agents \( a_1 \) and \( a_2 \). The manager \( M \) may block the data being sent by \( p_1 \) by simply attaching itself to the composite port of agent \( a_1 \) as shown in the figure. We will refer to the watch ring being used by the manager's obj-port \( p_3 \) as the *probe*. The manager's own obj-port \( p_3 \) will be in state \( O_{IR} \) at the time it attaches the probe to agent \( a_1 \). If \( a_1 \) has communication control over its parent clock ring and \( p_3 \) is in state \( O_{IR} \) at the time this probe
is attached, then as shown in row 2, and columns 2, 3 and 4, of Table 4.2, the composite port of \textbf{a1} will send an \textit{advance} signal to \textbf{p3} immediately after making the probe connection. If \textbf{a1} does not have communication control over its parent clock ring then it will be idle. In this case, control will reside with agent \textbf{a2}, who will transfer this control back to \textbf{a1} when agreement protocol is satisfied in its communication session with \textit{obj-port} \textbf{p2}. Agent \textbf{a1} will then send \textit{advance} to port \textbf{p3} of manager \textbf{M} once it gets back this communication control from \textbf{a2}. Thus, whether \textbf{a1} is active or idle at the time probe connection is established, it will always eventually send an \textit{advance} to \textit{obj-port} \textbf{p3} of the manager.

Once \textbf{p3} gets this advance the observer \textbf{M} may immediately get the data it is waiting for. No other communication events are necessary to enable \textbf{M} to get this data. Thus observation through probing will always satisfy the \textit{immediacy} property mentioned earlier.

A communication session in which \textbf{a1} is active will end only when \textbf{p3} also sends \textit{release} and/or \textit{trigger} signals needed to satisfy the agreement protocol. By withholding the \textit{release} signal, manager \textbf{M} may in this case prevent \textbf{p1}'s output data from being dispatched to \textbf{p2} and thus block \textbf{p2}'s parent object.

Thus data being delivered to a single object may be halted by intercepting its flow through the delivery side of a watch ring or by setting up a probe to the
composite port of an agent. Data being delivered to a group of obj-ports $G$ by an agent $A$ may be blocked by employing the technique illustrated in diagram C of Figure 6.1:

**Group Blocking with Observer Agent:** Here the manager $M$ will create an agent $A_1$ just before $A$ on the parent ring of agent $A$, and tune one of its own obj-ports to $A_1$ as shown in the diagram C. This will then enable $M$ to withhold data in a manner similar to the one described above.

**Group Blocking with Probe:** Group blocking may also be done by setting up a probe to a composite agent port, as described earlier.

One may notice, whereas blocking through an agent may be used not only to block data flow but also to examine the data that is being blocked, blocking through a probe will not enable a manager to examine data that is being blocked. Also, employing 'Port Blocking with Observer agent' one may prevent data from being delivered only to selected obj-ports in a group. Thus, selectively, one member of a group may be forced to stop computations in any selected activity cycle while others continue with theirs.

It is relevant here to see what would happen to members of a group some of whose input data have been thus blocked: Suppose data delivery was blocked to only one member, say $X$, of group $G$. This will not in any way affect computations in the other members of $G$, because as per our assumption only $X$ did not receive input data. However, it may be noted, agent $A$ will not forward outputs produced by these unblocked members of $G$ until $X$ also sends its output. This will assure that intended recipients of this data, i.e. those who are expecting to receive outputs from $G$, will not get incomplete data in which $X$'s contribution is not present. Thus, while halting one member of a group RESTCLK does not have to take any special actions to coordinate activities of the other unblocked members.

The important points to be noted in the above described blocking techniques are the following:
They are transparent to objects because no participation is required from them during establishment of the observation network. They have high resolution because every obj-port input/output can be observed separately. A manager's input/output itself can be observed by another manager. This satisfies the recursion requirement. Inputs to or Outputs from any member of a group can be selectively observed without interfering with any of the other members. While observation of data flowing on a ring will inevitably cause some delay, a manager observing this data can minimize this delay by sending a release immediately after receiving an advance and reading the accompanying data. This approach may be used to make the manager's agent $A_1$ forward the intercepted data to the next agent on the ring with minimal delay. This will contribute towards satisfaction of the minimal interference requirement.

Similarly these blocking techniques also satisfy all the properties of control discussed above: Minimal Disruption requirement is satisfied because only computations in the selected objects are halted. No a priori planning is required here since a manager can intercept and block data without requiring awareness of or collaboration from any of the objects in an application system. This satisfies the minimal planning requirement. Dynamic instantiation requirement is obviously satisfied since observation networks can be dynamically introduced at any time in RESTCLK. Selectivity requirement is satisfied since blocking can be used selectively on any specific obj-port in an application system.

6.3.2 In Situ Testing

A critically useful application, especially in large systems such as telephone and banking systems, is version testing. Here one is interested in testing a new version of an object in exactly the same network context in which the older version is being currently used. Further, we shall require that this should be done without
service interruption. We will refer to this as \textit{in situ} testing. The objective is to test the new version in the actual target environment in which it is intended to function. The hope is that such testing will guarantee more reliable validation than testing in synthetic or simulated environments. Also as we shall see, the \textit{in situ} testing facilities in RESTCLK allows for smooth substitution of old with new versions in functioning systems, again with no service interruption.

One possible \textit{in situ} testing scheme in RESTCLK is illustrated in Figure 6.4. The figure shows how a new version $O'$ of an object $O$ may be tested without interrupting services in a functioning system. The composite agent port of agent $A$, the agent at the top of the figure, supplies simultaneously the same input data, say $I$, to the old version, the new version and the comparator. This will happen in each activity cycle. The output data, say $D$, subsequently delivered by the old version is intercepted by agent $A_1$ and delivered to the comparator. Soon after recording its input $D$ the comparator will forward this data $D$ back to the agent $A_1$, who in turn will forward it to agent $A$. Thus, with minimal delay the output generated by the old version will reach the agent $A$. However, as we shall later see below, agent $A$ will be able to forward this data $D$ to its intended destination only after responses have been received from the comparator and agent $A_2$ (see figure).

The agent $A$ sends the input $I$ to the comparator \textit{via} one of its watch rings, as shown in the diagram. As soon as the comparator receives this input $I$ it will release the agent.

The output, say $D'$, generated by the new version to the same input $I$ is intercepted by the agent $A_2$ who also forwards it to the comparator. Immediately on receipt of $D'$ the comparator will release and trigger the agent $A_2$. It will not however send any data back to $A_2$. Thus, $A_2$ will have no data to forward to

\footnote{This is especially important in vital systems like telephone switching networks, banking networks, life support systems, power supply systems, etc.}
Figure 6.4: Illustrating An In Situ Testing Scheme in RESTCLK.
the agent \( A \) at the top of the figure. On receipt of release and trigger from the comparator, the agent \( A_2 \) will immediately pass on to agent \( A \) the control signals it had received from the new version \( O' \).

The comparator now will have the input \( I \) and the old and new version outputs \( D \) and \( D' \). It may process these in appropriate manners and send its outputs (for example, the results of comparison) to the agent \( A_3 \) at the bottom of the figure. \( A_3 \) will in turn pass the comparison results to its intended destination \( via \) the clock ring at the bottom of the figure.

Thus, for each input \( I \) the outputs \( D \) and \( D' \) may be compared to validate the new version. Validation data may be separately logged as needed and examined at a later time. Also, the outputs generated by the new version will not be seen by the system at any time during the validation process. If validation succeeds then the old version may be detuned from the system and the comparator and network components associated with it may all be removed. This will then cause the new version to be automatically integrated into the functioning system.

Clearly, the arrangement shown in the figure allows functionalities of new and old versions to be compared in identical network environments. It may be noted that the network augmentations needed to introduce and test the new version, and network changes needed for replacing the old version with the new, may all be done without in any way interrupting the services provided by the functioning system.

This is a general facility which may be used for all testing and validation during distributed systems development as well. However, in developmental testing multiple versions may not always exist. In such cases observation nets may be used to gather test data in order to validate them against specifications. We are not aware of any solution to this \textit{in situ} testing problem in any other communication paradigm that can allow such \textit{a priori} unplanned for strictly local \textit{in situ} testing without service interruption.
However, it should be mentioned that this testing and validation scheme is not failure proof. If the new version gets caught in a loop and does not send any outputs then the outputs generated by the old version will not get forwarded to their intended destinations. This can cause the entire system to hang. However, by using 'time-out's one may guard against such potentially catastrophic service interruptions.

All the desirable observation and control properties mentioned earlier, namely transparency, high resolution, recursion, minimal interference, minimal a priori planning, minimal disruption, high selectivity and dynamic instantiation are all satisfied in the above described in situ testing procedure. This can be easily verified.

6.3.3 Fault Management

Faults may be of two major types: System component failure and application component failure. A system component failure may involve a crash of one or more processes that implement RESTCLK. For such failures the conventional fault-management techniques such as, journaling, replication and shadow processing may be used for error recovery. Application component failure may involve a crash of one or more processes that implement the application objects. Handling of such failures will require detection of failure and taking corrective actions. We assume here that failure detection facilities will be a part of the application system, designed and incorporated during application system development. Corrective actions may involve system reconfiguration often involving replacement of failed objects with new versions, bringing the newly configured system to a consistent state, and restarting the system [28]. The support in RESTCLK for transparent dynamic reconfiguration provides a sound basis for managing faults in this manner. Again, it is easy to verify that actions taken for error detection and management will satisfy all the earlier mentioned desirable properties.
In the next chapter we discuss how RESTCLK facilities may be used to extend the security environment of application systems to the RESTCLK network in a manner such that application system security will be preserved over the network. We also show how active agents in RESTCLK may be used to provide special kinds of security monitoring facilities. Active agents in RESTCLK may also be used to selectively encrypt data in transit over designated parts of a network at specified time intervals. Parts of networks and intervals of time over which data is so encrypted may change dynamically during the life of an application system; this need not be planned for \textit{a priori} in an application system.
Chapter 7

Security in RESTCLK Networks

As we have already seen, the RESTCLK paradigm allows practically unrestricted access to data in transit. Also it makes it possible to make arbitrary dynamic changes in communication network structures. Thus RESTCLK provides new opportunities for breaking security. Therefore preserving the security environment of an application system in a RESTCLK communication network becomes an important issue.

RESTCLK is a network resource management system. As a resource management system, RESTCLK does not seek to set policies for security. It merely provides mechanisms at its lowest level that make security enforcement possible. It is instructive here to examine a bit the analogy with well known resource management systems, like database management systems [9] or operating systems [36]. These systems also do not set security policies. Instead provide abstractions in terms of which an application system designer may define security policies. They also provide mechanism which may be used to enforce security policies defined in terms of the said abstractions.

Taking analogy from this, RESTCLK defines abstractions for specifying security and provides enforcement mechanism in a security layer. Policies defined by application system developers in terms of the abstractions will then be enforced by the security layer. Defining such abstractions and developing security layers for enforcement are well established technologies [32, 10, 35, 7] and are outside the scope of this dissertation. Therefore we will not here enter into a discussion
of details on formats specific to various abstractions or structures and algorithms specific to security layer implementations. We will, however, discuss in some detail one specific approach to defining such abstractions and show how this abstraction may be used to uniformly extend application system security environment to a RESTCLK network. Also, we will discuss certain dynamic security monitoring capabilities which are made possible in RESTCLK.

The abstraction discussed here is based on the access control list \cite{2} method for specifying system security. It is assumed that all RESTCLK network objects and application system objects are classified into security classes. Class membership of an object will define the security status of the object: What the object may and may not do, what data it may access and with whom it may communicate. The association between RESTCLK network objects and security classes may be specified at the time of their creation and may optionally be modified afterwards. Application objects may associate security classifications also with messages they send. During the lifetime of a RESTCLK network each network object may accumulate information on dynamic security related attributes and remember these attributes in a security state. This security state may change with every activity cycle. Dynamic security monitoring functions may then be triggered into action based on the security state of designated objects. These are discussed in more detail in the following sections.

We will assume throughout that each handle will contain an entity ID and security classification associated with the entity. The dynamic security state corresponding to a handle might be accessed using the handle as a key.

\footnote{Such as counts of messages with various security classifications that were received and sent out by a network object, number of ring attachments/detachments and classifications of rings and ID's of agents who requested them, number of times an object communicated with another at a lower level of security, etc.}
7.1 The RESTCLK Security Layer

Figure 7.1 shows the security architecture in RESTCLK. The diagram shows a security subsystem which shares with RESTCLK its table of handles. Application objects are shown floating in the ambient operating system environment. The RESTCLK box itself is shown as containing several network agents. The network ring connections among these agents and between agents and application objects are not shown in the diagram. In the following we will always use the term *object* to refer to an application system object. The term *component* to a RESTCLK network component, which may be an obj-port, an agent or a representative. The term *entity* is used to refer to an application system object or a network component. RESTCLK will receive four kinds of requests from application objects:

1. create or destroy new entities,
2. establish or remove connections between existing entities,

3. change security classification of an entity,

4. change security compatibility of a pair of classifications.

As mentioned earlier, an object that requests creation or destruction of an entity will usually supply a unique identity for the entity to be created or destroyed. If an identity is not supplied with a creation request then RESTCLK will generate and assign one. The requester might also supply a *handle* for the new entity that is to be created. This handle may specify the security classification for the object. Then RESTCLK will supply this handle together with the handle of the requester to the *security layer* in order to obtain permission to create or destroy it. Based on the response obtained from the security layer RESTCLK will carry out the appropriate operations.

Installation of a newly created object within an application system will usually require specifications on how it should be connected to already existing system entities. This is the second kind of request mentioned above. In this case, the entities will already exist in the system and therefore RESTCLK will know their associated *handles*. A requester will provide the ID’s of entities to be connected and RESTCLK will request permission for the necessary operations by supplying to the security layer the ID’s of the entities to be connected and the ID of the requester. Security layer can access the handles associated with these ID’s from the shared handle table, shown in the figure. It will grant or deny permission based on the information in the handles and possibly also the dynamic security states associated with them.

Changing security classification of an entity is the third kind of request mentioned earlier. Again the security layer will make the determination to allow or disallow the requested change based on the information provided by the handle of the requester, the handles of the entities whose classifications are to be changed
and dynamic security states associated with them.

Requests to change security compatibility of given pairs of classifications is intended primarily for introducing new security classifications. If used to change compatibility of existing classifications, it may have major consequences in an application system. RESTCLK will accept such a change request only if it does not imply invalidation of any of the existing pathways.²

The purpose of discussion in this chapter is merely to show how facilities provided by RESTCLK may be used to provide a variety of security management schemes. We will discuss in some detail one particular security management scheme for extending application system security environment to RESTCLK network components.

### 7.2 A Security Management Scheme for RESTCLK

#### 7.2.1 Security Compatibility of Classifications

We assume here that for every entity the *Manager of Application System Security* (MASS) will assign to that entity a security classification. Let $S[X]$ be the classification assigned to $X$ by MASS. MASS will use the classifications to define a *security compatibility table*. This table will specify pairwise *security compatibility* between classifications. The security layer of RESTCLK will use this table to check whether an object with a given classification may communicate with another or not: One entity may communicate with another if their classifications are *security compatible* as per the *security compatibility table* of the application system. Hereafter, in this chapter we simply use the term *compatibility* to refer to *security compatibility*. Also, we will say two entities are *compatible* if their classifications are *compatible*.

---

²It would, however, be the security layer's responsibility to allow or disallow such changes.
It should be noted that the *security compatibility* relation is reflexive, but is not necessarily symmetric or transitive. Thus, \( X \) may be able to send data to \( Y \) while \( Y \) may not be able to send data back to \( X \). Also, among three objects \( X \), \( Y \) and \( Z \), \( X \) may be able to send data to \( Y \) and \( Y \) may be able to send data to \( Z \), but \( X \) may not be able to send data directly to \( Z \) without using \( Y \).

When a group of objects, \( G \), is serviced by one agent, \( A \), then there is need to determine a security classification \( S[G] \) for the group as a whole. We will view \( S[G] \) as being independent of the classifications associated with members of \( G \). This will be, of course, application dependent. By convention we will assume that \textit{Mass will always assign classification }\( S[G] \)\textit{ to the agent servicing this group }\( G \)\textit{ based on application requirements. Thus for the purpose of security enforcements RESTCLK will always view a communicating group as a single agent.}

### 7.2.2 Validity of RESTCLK Pathways

We will say that \( \langle X_1, X_2, \ldots, X_k \rangle \) is a \textit{pathway} if each \( X_i \) is a RESTCLK network component, i.e., an agent, a representative, or an obj-port. The pathway, \( \langle X_1, X_2, \ldots, X_k \rangle \) is \textit{valid} if and only if adjacent pairs are \textit{compatible} (i.e., security compatible). RESTCLK will allow only valid pathways to exist in its communication network.

Clearly, entities in RESTCLK, may communicate with each other if and only if valid pathways exist between them. It may be noted that this does not preclude two incompatible entities, \( X \) and \( Y \), from communicating with each other. Because they can communicate if a valid pathway \( \langle X, A_1, A_2, \ldots, A_m, Y \rangle \), \( m \geq 1 \) can be established. Thus incompatible entities cannot communicate only if no valid pathways are available.

---

\(^3\)A common practice in political mediations!

\(^4\)Thus it is possible for decisions made by a group to have a much higher (or lower) security classification than that of any member of the group.
We will say that two classifications, $S_p$ and $S_q$, are \textit{one-way incompatible} if there is no sequence of classifications, $\langle S_1, S_2, \ldots, S_m \rangle$, such that all of the following pairs are compatible: $(S_i, S_{i+1})$ for $1 \leq i < m$, $(S_p, S_1)$ and $(S_m, S_q)$.

$S_p$ and $S_q$ are \textit{two-way incompatible} if both $(S_p, S_q)$ and $(S_q, S_p)$ are one-way incompatible.

By making a pair of obj-ports two-way incompatible one may prevent for ever communication occurring between them. By making them one-way incompatible one may prevent communication in one direction. These are useful features to have in a security system, especially if classifications of entities in the system can be changed dynamically.

\subsection{7.2.3 Classification of Data}

At the request of its parent object, an obj-port may associate a security classification with its output data. This classified data may be made secure in several ways during its transit through the RESTCLK network. A few are outlined below:

- When an agent, $A$, receives such classified data from its previous agent or obj-port on a ring, it may selectively suppress the classified data from clients whose security classifications are incompatible with that of the data.

- If a RESTCLK component, $X$, receives classified data $D$ from another component $Y$ then $X$ may decide to encrypt the data $D$. This decision may depend on the classifications $S[X]$, $S[D]$, and $S[Y]$. The encryption might be such that only entities with classifications compatible with $S[D]$ will be able to decrypt it.

Thus classified data may be protected during transit in a RESTCLK network from unintended recipients.

It may also be noticed that this ability to dynamically encrypt data when necessary while data is in transit is a feature that is unique to RESTCLK. It is a
consequence of using active agents.

7.2.4 Time Varying Classifications

Classifications of entities in RESTCLK may change from one activity cycle to another. However, such changes may never be dependent on data in transit. RESTCLK will accept a change in security classification only if it does not invalidate any of the existing pathways.

It may be noted, changes that are not accepted by RESTCLK when presented serially may become acceptable when presented as a set of changes to be incorporated simultaneously. Also, the structure of RESTCLK network cannot be changed by changing classifications alone.

When data classification, described previously, is combined with changes in security classification of an entity, it may affect the set of recipients who might receive classified data from the entity or might affect eligibility of that entity itself to receive such data.\(^5\) By combining this feature with data classification, described previously, one may automatically secure data from being received by unintended recipients without having to change the structure of the RESTCLK network in any manner. This again is a feature that is unique to RESTCLK.

7.2.5 Dynamic Monitoring

RESTCLK allows certain kinds of dynamic monitoring of data while it is migrating over the RESTCLK network. When an entity \(X\), sends data \(D\) to a recipient \(A\), where \(A\) is an agent, one may have three associated classifications, \(S[X], S[D]\) and \(S[A]\). \(A\) will have knowledge of the above classifications and classifications of all of its clients. It will also have access to dynamic security states associated with...

---

\(^5\)Thus, it is possible that communication among members of a group is kept secret while negotiations are in progress, until it is decided that secrecy is no longer necessary.
with these entities. Based on conditions that depend on these classifications and
dynamic security states, A may take certain special actions to invoke application
specific daemons to log particular data transmission events or raise alarms on im-
pending security breaches. Records logged in this manner may consist of the data
$D$, handles of participating entities, time-stamps and other pertinent information.
A file of such records may be used to trace security breaches when they occur or
may be used to periodically check and validate system performance. This feature
is also unique to RESTCLK.

7.3 Concluding Remarks

We have presented in this chapter one possible way in which application system
security environment can be extended to a RESTCLK network by application
system developers. We have also discussed certain new ways of dynamic security
enforcement, and monitoring of classified data, that are unique to RESTCLK.

In the next chapter we will discuss details of RESTCLK implementation,
certain scalability and efficiency issues, possible extensions to RESTCLK.
Chapter 8
Implementation

RESTCLK has been implemented on several platforms over the last several years. The latest implementation (August 1997) was done in the WINDOWS NT environment using VISUAL C++. The previous versions were implemented on UNIX and on WINDOWS 3.1, using C. These implementations were used to produce several demonstrations illustrating RESTCLK applications. Some of these are discussed in Chapter 9. We discuss here the API for RESTCLK and analyze some of the implementation issues related to efficiency and scalability.

8.1 RESTCLK API

Context for Using RESTCLK API: System context in which an application system developer will use RESTCLK API is schematically shown in Figure 8.1 and Figure 8.2. For each application system object, its source code will be compiled with a library of routines provided by RESTCLK. This library is referred

Figure 8.1: Illustrating the Application System Object Compilation Process.
to in Figure 8.1 as the RESTCLK API library. This compilation process will generate the corresponding application object executable image. The purpose of the RESTCLK library routines is to interpret the API commands that may appear in the application object source codes. For each API command appearing in the source code the library routines will augment the command, where necessary, with appropriate run time parameters, for example identities of objects that request the API. Thus, requester identities are not included in command arguments in the API commands shown in Table 8.1. Figure 8.2 illustrates the relationship between the compiled application object images and the run time instances created from these images.

Table 8.1 shows the command formats for the RESTCLK API. These
<table>
<thead>
<tr>
<th>Data type</th>
<th>Returned Data</th>
<th>Command Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>int 0, 1</td>
<td>release_obj_port(obj_port_id)</td>
<td></td>
</tr>
<tr>
<td>int 0, 1</td>
<td>trigger_obj_port(obj_port_id)</td>
<td></td>
</tr>
<tr>
<td>int obj-port-id</td>
<td>look_for_advance(obj_port_ids.array, count, wait_flag)</td>
<td></td>
</tr>
</tbody>
</table>

| int 0, 1  | write_to_obj_port(obj_port_id, data, length) |
| int 0, -1, -2 | read_from_obj_port(obj_port_id, data, length) |

| int 0, 1  | create_object(obj_id, handle, exec_image_name, obj_port_ids.string, restclk_id) |
| int 0, 1  | create_clock_ring(clock_ring_id, handle, restclk_id) |
| int 0, 1  | create_watch_ring(watch_ring_id, agent_id, obj_port_id) |
| int 0, 1  | create_agent(agent_id, handle, ring_id, reference_agent_id) |
| int 0, 1  | create_obj_port(obj_port_id, handle) |
| int 0, 1  | destroy_object(obj_id) |
| int 0, 1  | destroy_clock_ring(clock_ring_id) |
| int 0, 1  | destroy_watch_ring(watch_ring_id) |
| int 0, 1  | destroy_agent(agent_id) |
| int 0, 1  | destroy_obj_port(obj_port_id) |

Table 8.1: RESTCLK Application Programmer Interface Commands.
commands are described below. It should be noted that in all the following operations the ID of the object that requests an operation does not appear as an argument in the API command format. As mentioned earlier, this is a runtime parameter that will be supplied by compiled codes of application objects, generated from RESTCLK API library. The commands only indicate the ID's of the pertinent objects. It may be noted, as mentioned earlier in Chapter 7, that handles associated with these objects will be retrieved or defined by RESTCLK or security layer, as appropriate. These handles may always be accessed using entity ID's as keys.

In synchronization and communication commands a parent object may refer to only the obj-ports that it owns. This restriction does not apply to configuration operations, except the create_obj_port operation. All other configuration operations may be requested by any object in an application system.

### 8.1.1 Synchronization Operations

**release_obj_port(obj_port_id):** This operation is used by a parent object to request the obj-port specified by obj_port_id to send out a release signal. A parent object should use this command only once in each activity cycle.

**trigger_obj_port(obj_port_id):** This operation is used by a parent object to request the obj-port specified by obj_port_id to send out a trigger signal. A parent object should use this command at most once in each activity cycle.

Since the obj-port will remember whether a release had been sent earlier or not in the current activity cycle, it will determine whether a pre-release trigger or a post-release trigger should be sent.

**look_for_advance(obj_port_ids_array, count, wait_flag):** This command is used by parent objects to poll the obj-ports in the specified obj_port_ids_array
for receipt of *advance*. The order in which the specified obj-ports are polled will be determined by the library routines in the RESTCLK API library. The *count* specifies the number of obj-ports in the array. The polling process will return control to the parent object as soon as an *advance* is sensed in any of the obj-ports in the array. The *wait_flag* will specify whether the polling process should wait until an *advance* is sensed at any of the obj-ports in the array, or whether it should return control immediately to the parent object if no *advance* is sensed.

### 8.1.2 Communication Operations

The read and write operations described below may be used only when an obj-port is in the active period of an activity cycle.

**write_to_obj_port**(obj_port_id, data, length): This command is used by a parent object to instruct the obj-port specified by the obj_port_id to write output data into the designated private memory of the port. *Data* is pointer to the actual data to be written into the designated memory and *length* is the length of this data.

**read_from_obj_port**(obj_port_id, data, length): This command is used by a parent object to instruct the obj-port specified by the obj_port_id to read input data from the designated private memory of the agent tuned to it. Data in this designated memory is assumed to be a collection (*bag*) of individual data items. Each invocation of this command will read one item from this bag. *Data* here is a pointer to the location where the input should be stored and *length* is the *return parameter* that is used by the command to return the length of input data item that was read.

After the last data item in the bag had been read, this will return '-1' to indicate that this was the last data item in the bag. If the bag was empty at the time this read operation was invoked then it would return '-2'.
8.1.3 Configuration Operations

In all the create operations below the handle parameter in the argument list will indicate the requester specified security classification, if any, for the new entity to be created. If the requested classification is not acceptable to the security layer, then it will deny permission for the creation of the entity. If requester does not specify a security classification then a default classification will be assigned by the security layer.

create_object(obj_id, handle, exec_image_name, obj_port_ids_string, restclk_id): This is used to create a new instance of an executable image of an object. The image to be used for creating the instance is specified by exec_image_name. Obj-id is the ID of the new instance. The list of obj-port to be created for the new instance is specified by obj_port_ids_string. The restclk_id is the ID of the RESTCLK manager in whose domain the new object should be created.

create_clock_ring(clock_ring_id, handle, restclk_id): This will cause a new clock ring with the specified ID to be created. The creation of this clock ring will also cause its primary agent to be created and installed for it. This primary agent ID will be the same as the clock ring ID. The restclk_id will specify the ID of the RESTCLK manager in whose domain the new clock ring should be created. The initial state of the primary agent’s composite agent port will be set to $C_{tr}$. This will prevent it from sending an advance to itself via its ring port.

create_obj_port(obj_port_id): As mentioned earlier only a parent object may request an obj-port to be created for it. If successful this operation will cause a handle to be defined for the new obj-port and a private memory to be designated for the port. The initial state of this obj-port will be set to $O_{tr}$.

create_agent(agent_id, handle, ring_id, reference_agent_id): This specifies the agent id for the new agent, the ring (clock or watch) to which this agent should be attached and the reference location for the agent on its parent ring. As
mentioned earlier the reference location will specify the ID of the agent that will be just before the new agent in the parent ring. The state of the newly created agent will be set to $C_{TR}$.

`create_watch_ring(watch_ring_id, handle, agent_id, obj_port_id)`: This will create a new watch ring connecting the specified obj-port and the composite agent-port of the specified agent. The operation will be successful, of course, only if all the security and state compatibility conditions discussed earlier in Chapters 7 and 4 are satisfied.

It should be noted that in the following destroy operations, any time an entity gets destroyed its associated `handle` will also get destroyed.

`destroy_object(obj_id)`: An object can be destroyed only if it is totally isolated from the RESTCLK network, i.e., none of its obj-ports should have any watch rings attached to them. As a part of this operation all the obj-ports of the object will also get destroyed.

`destroy_clock_ring(clock_ring_id)`: A clock ring may be destroyed provided the only agent attached to the clock ring is its primary agent and this primary agent does not have any watch rings attached to it. When the clock ring is destroyed the primary agent will also get destroyed.

`destroy_watch_ring(watch_ring_id)`: A watch ring may be destroyed at any time provided that no watch agents are attached to it. Watch ring creation and destruction processes were discussed in detail in Chapter 4.

`destroy_agent(agent_id)`: An agent may be destroyed only if no watch ring is connected to its composite agent port. This operation may not be used on a primary agent of a clock ring.

`destroy_obj_port(obj_port_id)`: Only a parent object may request one of its obj-ports to be destroyed and this port may be destroyed only if it is not presently connected to a watch ring.
In the next section we discuss certain aspects of issues pertaining to communication efficiency and scalability.

8.2 Implementation Issues

8.2.1 Scalability

We will discuss here the following RESTCLK scalability issues:

1. Network Scalability,

2. Ring Scalability,

3. Group Scalability, and

4. Distribution Scalability.

RESTCLK Network Scalability: RESTCLK installation may consist of multiple instances of RESTCLK managers running on either a single machine or in multiple machines across a distributed network. Each such manager will have a unique identity. Identities of RESTCLK managers are used in two operations, create_object and create_clock_ring, described above. These two configuration operations will usually be used only by application system managers. Application system objects themselves need not know the individual RESTCLK manager identities. This is facilitated in the current implementation of RESTCLK by the use of a common name server.¹

RESTCLK does not put any restriction on the number of instances of RESTCLK managers a RESTCLK system installation may use. It also does not put any restriction on how the set of all RESTCLK entities in an application system

¹This name server may itself be distributed over a network. Use of single name server for all instances of RESTCLK managers eliminates the need for application system objects to be aware of identities of RESTCLK managers.
should be partitioned and assigned to the RESTCLK managers. Since there is no
limit on the number of instances of RESTCLK managers an application system
may use, it is always possible to restrict the number of RESTCLK entities as-
signed to any one manager without restricting the number of RESTCLK entities
in the whole application system. Thus, RESTCLK network size may be scaled up
arbitrarily in an application system without significant performance degradation.

RESTCLK Ring Scalability: RESTCLK by itself does not put any restric-
tions on the number of agents that may be attached to a ring. However it may
be noted, each addition of an agent to a ring will introduce inevitable delays in
object to object (or group to group) communication. Thus, the number of agents
that may be introduced into a ring will be limited by application performance
requirements.

RESTCLK Group Scalability: Here again RESTCLK does not put any
limits on the size of communication groups or distribution of members of a group
across machines. As shown in Figure 2.9 members of a communication group
may reside anywhere in a distributed system. By delegating responsibilities to
representatives organized in a hierarchy, RESTCLK may limit the number of obj-
ports tuned to any one composite agent port without limiting the total size of a
communication group. It may also be noted that use of hierarchies of represen-
tatives minimizes cross network communications. Thus one obtains scalability of
communication group size in RESTCLK networks. However one will never be
able to avoid increases in delays associated with group communication as group
size increases.

RESTCLK Distribution Scalability: RESTCLK puts no limits on the
extent to which a distributed application system may be dispersed over geo-
graphical regions, since each computer in the distributed system may have its
own instances of RESTCLK manager and communication across networks will all
occur only between RESTCLK agents and their representatives and not between
individual senders and receivers. This will minimize the number of cross network message exchanges that occur. Also, communication delays added by RESTCLK to cross network communications will be negligible compared to intrinsic network delays themselves. Thus, the extent to which an application system may be distributed will be limited only by the application system performance requirements and not by requirements of RESTCLK or communication delays introduced by RESTCLK.

Analysis of RESTCLK communication delays discussed in the next subsection compares delays in RESTCLK with delays in other existing modes of communication. Analysis shows that even though, at times, RESTCLK may introduce more communication delays in certain modes of communication than other communication schemes, the overall performance of RESTCLK is comparable to other schemes that are currently in use. RESTCLK communication is particularly effective in group to group communication, a feature that is unique to RESTCLK.

8.2.2 Communication Efficiency

Our objective here is to show that RESTCLK adds little or no overhead in terms of communication delays compared to any other agent mediated communication system. To establish this we analyze communication delays in port to port, port to group, and group to group communication. In each case we show that communication delays in RESTCLK are either comparable or better than other agent mediated communication systems.

Clearly RESTCLK has *indirection* in its communication architecture to support flexibility. As pointed out by Segal [41] without some form of indirection flexibility is impossible to achieve. Thus, it is reasonable to compare RESTCLK only with other agent mediated systems.

We will analyze and compare delays in three modes of communication: (i) Direct multicast (ii) Agent mediated multicast and (iii) RESTCLK. These are
illustrated in Figure 8.3. In direct multicast, shown at the top of the figure, each sender multicasts its output message to all the intended receivers. In agent mediated multicasting, shown at the center, each sender sends its output message to an agent. Agent is responsible to multicast each output message to all the intended recipients. In RESTCLK, shown at the bottom, two agents are involved: The first gathers together in memory $M_A$ output messages sent by all senders. The second forwards the collection of all messages in $M_A$ simultaneously to all the recipients.

Let $T[n, m, s]$ denote the total time taken per communication session for $n$ senders to send a message $M$ of size $s$ to $m$ receivers. We will first do an approximate analysis of the delay $T[1, m, s]$ for each scheme, for the case where there is only one sender. For each multicast we will assume that $T[1, m, s]$ may be expressed in terms of the following time delays:

- $T_{\text{write}}$: Time for a writing a message of size 1.
- $T_{\text{notify}}$: Time for notifying a receiver.
- $T_{\text{read}}$: Time for reading a message of size 1.
- $T_{\text{ack}}$: Time for acknowledging receipt of message.

We assume shared memory communication throughout, and all senders and receivers will know a priori the address of the shared memory. It should be emphasized that the analysis presented below is only approximate. It is however adequate for the purpose of comparisons discussed below.

In direct communication,

$$T_{\text{direct}}[1, m, s] = sT_{\text{write}} + mT_{\text{notify}} + \alpha msT_{\text{read}} + T_{\text{ack}}.$$  

This assumes notification is done sequentially $m$ times to the $m$ receivers. Reading of message of size $s$ by $m$ receivers is assumed to be $(\alpha m \times s)T_{\text{read}}$ where $0 < \alpha \leq 1$ is a constant of proportionality. Here we assume arbitrary overlapping of reading with the sequential notification process, i.e. some receivers might be
DIRECT: Each of n Senders send one Multi-cast to m recipients.

AGENT: Multi-cast Agent does n multicasts to m recipients.

RESTCLK: Group to Group Communication.

Figure 8.3: Illustrating Different Communication Schemes.
reading the message while others are still being notified. The constant of proportionality $\alpha$ may be adjusted to account for overlapping between concurrent reading and notification processes. Each receiver will acknowledge receipt of message separately. The use of single $T_{ack}$ above may be understood as the time for receipt of the last acknowledgement from the last receiver; the previous ones are here assumed to overlap with reading and notification time, i.e. one receiver may send its acknowledgement while another is reading its message. Next message may be sent to a receiver only after this last acknowledgement has been received.

In agent communication,

$$T_{agent}[1, m, s] = (T_{direct}[1, 1, s] + T_{direct}[1, m, s]).$$

Here the sender first sends its message to the agent. The term $T_{direct}[1, 1, s]$ above represents the time taken for this. Then the agent multicasts this message to the recipients. The second term above represents the time needed to do this. Thus agent mediated communication is in this case more expensive than direct communications.

One may assume that $T_{restclk}[1, m, s] = T_{agent}[1, m, s]$ even though RESTCLK requires two agents to complete the same communication. This is because when two RESTCLK agents communicate, all they do is transfer communication control over the clock ring. They do not have to write and read the message again. Thus the message already written in memory $M_A$ shown in the figure, will simply be forwarded by the second agent to its clients, when it takes control of communication. At best this may require one extra notification. However, if both agents are implemented in the same process, then this notification time can be ignored.

Thus one may conclude that multicast communication from one sender to multiple receivers would cost in RESTCLK no more than what it would cost in any agent mediated communication. Thus with little or no increase in cost, RESTCLK
provides all the significant features of flexibility and dynamicity described earlier. Of course, if observers are introduced in a communication path it will increase associated delays. This is inevitable. However, this additional delay will exist only when this additional observer functionality is added to an application system, not otherwise.

The significant feature of RESTCLK communication that is unique to RESTCLK is, of course, the group to group communication facilities that the other schemes do not support. Communication delay for $n$ senders to multicast their respective messages to $m$ receivers using direct communication will be

$$ T_{direct}[n, m, s] = nT_{direct}[1, m, s], $$

since separate communication should occur $n$ times, once for each sender. In agent mediated communication the total time delay may be assessed as follows:

$$ T_{agent}[n, m, s] = sT_{write} + T_{notify} + n(sT_{read} + sT_{write} + mT_{notify} + \alpha msT_{read} + 2T_{ack}), $$

where $sT_{write}$ is the time taken for the $n$ senders to write in parallel into their, respective, private memories messages of size $s$ that each is sending and $T_{notify}$ is the time taken for the $n$ senders to notify the agent, again in parallel, that messages are ready to be sent. The rest of the expression on the right is repeated $n$ times once for each sender. This includes time taken by agent to read message from a sender’s private memory ($sT_{read}$), time taken by agent to write this message into its own private memory ($sT_{write}$), time taken to notify the $m$ recipients in sequence ($mT_{notify}$), time taken by the $m$ recipients to read their common message from the agent’s memory in pseudo-parallel ($\alpha msT_{read}$), time taken to send acknowledgement in parallel to agent and the agent in turn to forward this acknowledgement to the sender ($2T_{ack}$). This expression reduces to,

$$ T_{agent}[n, m, s] = (n + 1)sT_{write} + (1 + nm + 2n)T_{notify} + (1 + \alpha m)nsT_{read}. $$

We have assumed here that no message pipelining occurs in the multicast message
delivery; i.e. agent will not multicast the next message before receiving acknowledgement for the current one.

Under the same assumption, in RESTCLK communication,

\[ T_{\text{restclk}}[n, m, s] = sT_{\text{write}} + T_{\text{notify}} + n sT_{\text{read}} + n sT_{\text{write}} + m T_{\text{notify}} + \\
+ \alpha mnsT_{\text{read}} + T_{\text{ack}} + n T_{\text{ack}}. \]

The sequence of events contributing to the above delays may be itemized term by term as follows

1. \(sT_{\text{write}}\): Assuming that the \(n\) senders write their respective output messages in their private memories in parallel it will take \(sT_{\text{write}}\) amount of time.

2. \(T_{\text{notify}}\): Notification of completion of writing by senders is also done in parallel. The \(T_{\text{notify}}\) time above represents the time taken for this.

3. \(nsT_{\text{read}}\): The messages in the private memories of senders are read by the agent in sequence. This takes \(nsT_{\text{read}}\) amount of time.

4. \(nsT_{\text{write}}\): To put all messages together the agent will write each message into its private memory, the memory \(M_A\) shown in the figure. This takes time \(nST_{\text{write}}\).

5. \(mT_{\text{notify}}\): The first agent in the figure will now transfer communication control to the second agent. This takes no time. The second agent will notify its \(m\) clients about the availability of message in \(M_A\). The notification will be done sequentially, taking time \(mT_{\text{notify}}\).

6. \(\alpha mnsT_{\text{read}}\): The \(m\) recipients may read the message in \(M_A\) pseudo-parallel: \(\alpha\) in \(\alpha mnsT_{\text{read}}\) accounts for this pseudo-parallelism; \(ns\) here is the size of data bag in memory \(M_A\), assuming data of size \(s\) for each of \(n\) senders.

7. \(T_{\text{ack}}\): This is the time taken by the recipients to notify the agent 2 that receipt of message has been completed.
8. \( nT_{ack} \): This is the time taken by agent 1 to notify \( n \) senders in sequence that next communication session may begin.

Here one may assume that \( T_{ack} = T_{notify} \) and simplify the expression above to,

\[
T_{restclk}[n, m, s] = (n + 1)sT_{write} + (m + n + 2)T_{notify} + (1 + \alpha m)n sT_{read}.
\]

Clearly,

\[
T_{direct}[n, m, s] > T_{agent}[n, m, s] > T_{restclk}[n, m, s].
\]

Thus in group to group communications RESTCLK has a clear advantage.

The above analysis shows that one is able to obtain all the benefits of RESTCLK without significant loss of performance. In its current version RESTCLK does not use data pipelining around clock rings. This, of course, may cause application object to waste time during idle periods of activity cycles. In the next section we present a possible enhancement to RESTCLK that uses data pipelining. Here, as mentioned earlier, multiple agents of a clock ring can be simultaneously active and objects need not waste time during idle periods of activity cycles. This can improve application system performance without loss of communication efficiency.

### 8.3 Possible Extensions to RESTCLK

#### 8.3.1 Pipelining

In the RESTCLK paradigm discussed so far in each clock ring at any time only one agent can be active. An activity cycle associated with an agent \( A_i \) is shown in Figure 3.1. Communication between the composite agent port \( X \) of \( A_i \) and an obj-port \( Y \) tuned to \( X \) in the figure can take place only during the active period of such an activity cycle. During the idle period of the activity cycle obj-port \( Y \) cannot communicate with \( X \). If the parent object of \( X \) wants to send out
more data through $X$ during such an idle period it will have to wait for the next activity cycle to begin. This wasted waiting time can be reduced considerably by the pipelining extension to RESTCLK, described below.

Let $A_1, A_2, \ldots, A_n$ be the agents around a clock ring with primary agent $A_1$. Let $\tau$ be a token associated with the clock ring. At any time let us suppose that only one agent can be the owner of this token. The agent that owns the token will be the active agent on the ring. If there is only one token, then only one agent can be active. We will introduce pipelining my associating multiple tokens with each clock ring. Let

$$T = [\tau_1, \tau_2, \ldots, \tau_k]$$

for $t > 1$ be the sequence of all tokens associated with a clock ring. We will say an agent $A_i$ on the ring is active at any point of time if and only if $A_i$ is holding a non-empty subsequence of these tokens. Activity around a ring at any time may then be represented as shown in Figure 8.4 where nine tokens have been mapped on to four agents on a clock ring. Here agents $A_1$, $A_2$ and $A_3$ are simultaneously active. $A_4$ is not active. $A_1$ is holding the subsequence of tokens $\tau_5$, $\tau_6$, $\tau_7$, $A_2$ is holding $\tau_1$, $\tau_2$, $\tau_3$, $\tau_4$ and $A_3$ is holding $\tau_8$, $\tau_9$.

Let $A_1$ be the primary agent of the clock ring. When the clock ring and the agent was created $A_1$ would have been assigned the sequence of all the nine tokens, $\tau_1$ through $\tau_9$. Subsequently at some point of time let us assume that the distribution of tokens around the ring reached the configuration shown in figure 8.4. To understand how the initial configuration in which all tokens were with agent $A_1$ may subsequently lead to the configuration shown in the figure, it is necessary to understand how client/agent interactions occur in the multi-token case. This is described below.

Let us consider interactions between agent $A_1$ and its clients in the configuration shown in the figure. As mentioned earlier $A_1$ has the subsequence of tokens
Figure 8.4: Illustrating Mapping of Subsequences of Tokens to Agents on a Clock Ring.
\(\tau_5, \tau_6, \tau_7\), and let us assume its composite agent-port, \(X\), services four client obj-ports, \(p_1, p_2, p_3\), and \(p_4\) as shown in the figure.

At this point \(\tau_5\) is the first token agent \(A_1\) has in its possession. Thus, interaction between \(X\) and the client ports it services will begin with \(X\) sending to its clients the \((\text{advance}, \text{token})\) pair, \((a, \tau_5)\) that corresponds to the first token \(\tau_5\). When clients receive this pair they will know that they can read input \(I_\tau\) corresponding to token \(\tau_5\) from the designated memory \(M_A\) of the agent. At this point, processing of \(I_5\) by the clients and interactions with the composite port \(X\) will all take place exactly as in the single token case.

For convenience let us recapitulate here the following signals used by the CPSM in group to group communication:

\[
\land r = (r_1 \land r_2 \land \ldots \land r_n), \\
\lor t = (t_1 \lor t_2 \lor \ldots \lor t_n),
\]

where \(r_i\) and \(t_i\) are the release and trigger signals shown in Figure 3.6. In the single token case each activity cycle was uniquely associated only with one token and at any point of time all clients had to be in the same activity cycle. Therefore it was not necessary to remember the token associated with an activity cycle.

In the multi-token case the situation is different. First of all, it is necessary to associate tokens with activity cycles, since a composite agent will have to keep track of more than one activity cycle at the same time, one corresponding to each token in its possession. Secondly, it may be noted that although each client will be in the midst of an activity cycle corresponding to a single token, different clients may be in different activity cycles corresponding to different tokens.

This requires some modifications in the composite CPSM and its associated agreement protocol network (AP-net) shown in Figures 3.5 and 3.6. Let us refer to the new multi-token CPSM as MCPSM, and the multi-token AP-net as MAP-net. The structure of MCPSM and MAP-net may be visualized as follows:
Mcpsm will have one set of four states for each token. Let us refer to the Cpsm corresponding to token \( \tau_j \) as \( \text{Cpsm}^j \). Transitions between the states of \( \text{Cpsm}^j \) will be driven by signals received from Map-net that corresponds to the token \( \tau_j \). Thus we will assume that the Map-net will have one set of \textit{release} and \textit{trigger} registers for each token: token \( \tau_j \) will have registers \( (r_1^j, r_2^j, \ldots, r_n^j) \) and \( (t_1^j, t_2^j, \ldots, t_n^j) \) associated with it, and the Map-net will generate one pair of signals,

\[
\land r^j = (r_1^j \land r_2^j \land \ldots \land r_n^j),
\]

and

\[
\lor t^j = (t_1^j \lor t_2^j \lor \ldots \lor t_n^j),
\]

for each \( \tau_j \). These signals will drive \( \text{Cpsm}^j \) in the Mcpsm. To do this, Map-net should have the ability to direct release and trigger signals received from client obj-ports to appropriate registers based on the tokens associated with the clients.\(^2\)

Let us now see how it may come about that different clients of the same composite agent \( X \) may happen to be in different activity cycles at the same time.

As stated earlier, the composite port \( X \) will begin its activities by first transmitting to its client obj-ports the signal \((a, \tau_5)\). Thus, all its clients will in turn begin processing the data \( I_5 \) that corresponds to \( \tau_5 \). Different clients may take different amounts of time to complete processing this data. Thus client \( p_4 \) may complete processing before the other clients. In this case release signal \( r_4 \) associated with token \( \tau_5 \) may reach Map-net before other release signals. If the Map-net had not received any trigger corresponding to \( \tau_5 \) from any of its clients, even though \( p_4 \) had completed its processing of \( I_5 \), agent \( A_1 \) will not forward to it the next \((\text{advance}, \text{token})\) pair \((a, \tau_6)\). However, as soon as \( \lor t^5 \) becomes true \( A_1 \) will send \((a, \tau_6)\) to each client \( p_j \) from whom Map-net had already received the release signal \( r_j^5 \). Thus, it is possible that \( p_4 \) can progress to \( \tau_6 \) activity cycle

\(^2\)It is not hard to realize this. For example, one could use hardware or software \textit{demultiplexers} to implement this.
while others are still in the $\tau_5$ cycle. Thus depending on the relative speeds with which different clients process their input data they may be in different activity cycles.

When agreement protocol is satisfied for $\tau_5$ the MAP-net will clear all the $\tau_5$ registers and the agent $A_1$ will forward to its next agent $A_2$, in Figure 8.4, the collection of $\tau_5$ outputs, say $O_5$ received from all its clients by sending to $A_2$ the pair $(a, \tau_5)$. This will cause the token to migrate from $A_1$ to $A_2$. The important point to note here is as the various tokens migrate around the clock ring their initial order will always be preserved, i.e., tokens will never overtake one another. Through repeated migrations of tokens in this manner from one agent to another, one can now see how the initial configuration in which all tokens were with $A_1$ may at some time result in the configuration shown in Figure 8.4.

This completes how RESTCLK protocols may be extended to enable pipelining using multi-tokens without requiring any modifications to obj-port structures. We have not here discussed how application objects may keep track of their inputs and outputs when pipelining is allowed. In the single token case, each application object would know when it received an input the prior output, if any, to which the received input corresponded. In multi-token case an application object may send a sequence of outputs before it receives a response for its first output. Let us refer to such a sequence of outputs, for which no corresponding inputs have yet been received as the pending requests of the object. In the single token case each object will have at most only one pending request at any time. In the multi-token case there can be more than one pending requests. Thus application objects should have some facilities available to them to establish correspondences between inputs they receive and prior outputs in the pending request queue. The pipelined RESTCLK tokens may be used by application objects to establish and maintain this correspondence. However, it may not be possible to provide an application
independent RESTCLK solution to the general problem of request/response correspondence maintenance in dynamically changing pipelined systems. This is at present an open problem.

8.3.2 Open Problems

We will conclude this chapter by briefly outlining a few other open problems and research issues that pertain to RESTCLK implementation. One problem that was mentioned earlier in Chapter 4 is the problem of implementing RESTCLK in its own paradigm. In other words implementing RESTCLK in a manner such that all of RESTCLK's own communications with its network components will become available for observation and control, in the same manner as application object communications. Even an operating system may then be implemented using RESTCLK paradigm. We believe these are possible. Further investigations are needed to achieve these objectives.

One may build hardware devices that implement RESTCLK using hardware chips containing arrays of RESTCLK port sequential machines of various kinds. Just as automatic memory allocation and memory management is used in conventional compilers, one may use automatic port sequential machine allocation and management to compile communication interfaces as and when needed. Software implementations of RESTCLK impose an intrinsic limit on the number of concurrent communication processes that can be active at any one time. Hardware implementations of RESTCLK can provide considerably more concurrency than software implementations can. The design issues and architectures for such RESTCLK implementation require further research.

In the next chapter we present a few representative applications that illustrate how the power of O&C in RESTCLK, and RESTCLK support for simultaneous data delivery, may be exploited for dynamic reconfiguration of application systems.
Chapter 9

Examples of RESTCLK Applications

In this chapter we will present a few examples, which were actually implemented and demonstrated in various versions of RESTCLK. These examples illustrate the use of observation and control for transparent dynamic reconfiguration of application systems in various situations. We also use these examples to compare RESTCLK solutions for dynamic reconfiguration with solutions proposed in other systems like POLYLITH [19, 38, 21, 22, 20, 5] and CONIC [25, 27, 26, 24]. These two are the only other significant systems that we are aware of, which propose novel communication systems organization that facilitate dynamic system reconfigurations of the kinds discussed here. The examples discussed in this chapter will illustrate clearly the benefits gained by using RESTCLK.

The first example compares the RESTCLK solution to the dynamic repartitioning problem in parallel/distributed Jacobi relaxation algorithm [3], studied by Hofmeister [19] with a POLYLITH solution. The second example compares the RESTCLK solution to the Evolving Philosophers problem introduced by Kramer and Magee [25] with both CONIC and POLYLITH solutions.

A third example was implemented and demonstrated in an earlier version of RESTCLK to demonstrate the use of simultaneous delivery in RESTCLK. This facility was used to implement a distributed BLACKJACK game with players located at different machines. The RESTCLK implementation allowed the number of players in a game to change dynamically while the game was in progress and yet did not require a new source code for the players. We are not aware of
other implementations for this game that offer similar capabilities. Details of this example are however not discussed in this chapter.

The most significant features of RESTCLK solutions illustrated by the examples discussed here are the following:

1. **Minimal a priori planning**: In all examples discussed here, systems that were changed dynamically using RESTCLK solutions, did not require prior planning in their design in order to accommodate such changes. This is because RESTCLK required no collaboration from application systems in order to change them as and when needed. This is a unique feature of RESTCLK. Solutions in other systems like POLYLITH and CONIC require considerable collaboration from application systems in order to do such changes.

2. **Reduced Code Complexity**: Since implementations that use RESTCLK do not have to anticipate and accommodate all possible future changes that may occur, application object source code will not need to embed within them logic needed for accommodating dynamic changes. This results in considerably reduced object source code complexity.

3. **Enhanced Reusability**: Since in RESTCLK application object source codes can be independent of network contexts in which they might be used during their lifetimes, they can be reused in a variety of network context during their lifetimes. Thus they will have enhanced reusability.
Dynamic System Replacement in Jacobi Relaxation

9.1.1 Problem Statement

It is often necessary to dynamically replace a system of interacting objects with a new system with minimal service interruption of system operations, while preserving system functionality. In order to preserve functionality, we shall require that it should be possible to stop a running system dynamically at any one of its consistent states, reconfigure it as needed to arrive at its new configuration and restart the new system from the same consistent state. This would eliminate the need to restart the new system again from its initial state or recompute any of its previously computed results.

We will solve this general problem in the context of a parallel/distributed implementation of the Jacobi Relaxation Technique [3] for solving partial differential equations. The problem here may be stated as follows: As shown in Figure 9.1 sequential implementation of Jacobi Relaxation Technique will involve initializing a large matrix and then repeating a matrix computation for a set...
number of iterations.

The parallel/distributed version is obtained by partitioning an \((M \times N)\) matrix, \(S^0\), into \((m \times n)\) submatrices \(S^0_{ij}\) for \(1 \leq i \leq m\) and \(1 \leq j \leq n\), and assigning each submatrix to a separate process. We will refer to each such process as a worker. Workers are organized in an \((m \times n)\) grid pattern as shown in Figure 9.2. Let \(W_{ij}\) denote the worker in row \(i\) and column \(j\) of the grid and let \(S^0_{ij}\) be the submatrix initially assigned to \(W_{ij}\) as shown in the figure. The relationship between the submatrices and the total matrix is expressed by,

\[
S^0 = \left[ S^0_{ij} \right]_{1 \leq i \leq m, 1 \leq j \leq n}.
\]

Figure 9.2 illustrates the case for \(m = 4\) and \(n = 3\).

Using the respective submatrices \(S^0_{ij}\) the \((m \times n)\) workers in the grid will together carry out computations that are equivalent to the sequential computations on the total matrix \(S^0\) performed by the flow chart shown in Figure 9.1. At the beginning of each iteration each worker will exchange with its neighbors certain boundary values. In iteration \(g\) the boundary value transmitted by worker \(W_{ij}\)
Figure 9.3: Flow-chart of Worker Algorithm.

will be determined by the current value of the submatrix, say $S_{ij}^g$, at the worker after $g$ iterations. The total matrix after $g$ iterations will be,

$$S^g = \left[ S_{ij}^g \right]_{1 \leq i \leq n, 1 \leq j \leq n}.$$

The flow chart of the worker algorithm is illustrated in Figure 9.3.

Evolution of the system may involve dynamic repartitioning of the matrix $S^g$ after $g$ iterations, from $(m \times n)$ submatrices to $(m' \times n')$ new submatrices,

$$S^g = \left[ U_{ij}^g \right]_{1 \leq i \leq m', 1 \leq j \leq n'},$$

reconfiguring the worker grid from $(m \times n)$ workers to a new grid of $(m' \times n')$ workers, and assigning to each worker its corresponding new submatrix as illustrated in Figure 9.4, for the case $m' = 3$ and $n' = 2$. Here $W_{ij}'$ denotes the workers in the new grid. These new workers will also be running the same algorithm shown in Figure 9.3. However the submatrices $U_{ij}^g$ used by worker $W_{ij}'$ in the new configuration will be different from the submatrix $S_{ij}^g$ used by worker $W_{ij}$ in the old configuration. The submatrices in the old and the new configurations satisfy the following relationship:

$$S^g = \left[ S_{ij}^g \right]_{1 \leq i \leq m, 1 \leq j \leq n} = \left[ U_{ij}^g \right]_{1 \leq i \leq m', 1 \leq j \leq n'}.$$
For the examples shown in Figures 9.2 and 9.4 we will thus have

$$S^g = \left[ S_{ij}^g \right]_{1 \leq i \leq 3}^{1 \leq j \leq 3} = \left[ U_{ij}^g \right]_{1 \leq i \leq 3}^{1 \leq j \leq 3}.$$

This condition has to be satisfied in order to continue computations with the new configuration.

To do this we will make use of the concepts of system states and consistency of system states.

**Definition:** System State at any time $t$ is the collection of submatrices, $S_{ij}^{x(i,j,t)}$, for $1 \leq i \leq m$ and $1 \leq j \leq n$.

The superscript $x(i,j,t)$ in $S_{ij}^{x(i,j,t)}$ here is the local iteration number associated with worker $W_{ij}$ at time $t$. This indicates that this worker has performed exactly $x(i,j,t)$ iterations as of time $t$. It may noted, at any point in time the local iteration numbers of workers in a grid may not all be the same. However, worker algorithm is such that local iteration numbers of any two neighboring workers cannot differ by more than 1. Thus, in a $(4 \times 3)$ grid shown in Figure 9.2, the difference between the local iteration numbers of workers $W_{11}$ and $W_{43}$ can be at most 5, as shown in Figure 9.5. In general for an $(m \times n)$ grid of workers this can be at most $(m + n - 2)$. We will say,
Figure 9.5: Illustrating Possible Local Iteration Numbers of Workers.

**Definition:** A System State is **consistent** at time $t$ if and only if
\[ \forall (i) \forall (j) (x(i, j, t) = k), \] where $k$ is any integer constant.

One can now claim,
\[ S^g = \left[ S^{x(i, j, t)}_{ij} \right]_{1 \leq i \leq m} \]
if and only if $\forall (i) \forall (j) (x(i, j, t) = g)$. Thus, at any point in time the system state will be consistent if and only if all workers have completed exactly the same number of iterations as of that point in time.

In order to dynamically reconfigure a working system one has to then do the following:

**Tasks to be performed for reconfiguration:**

1. Bring the system to a consistent state at some point $t$ in time and halt it.
2. Construct the system state at time $t$ by retrieving $S^{x(i, j, t)}_{ij}$ from each worker and use these to build the total matrix $S^g$ as shown in the above equation.
3. Destroy the current worker grid configuration.
4. Construct the new submatrices $U^g_{ij}$.
5. Create the new worker grid with workers $W'_{ij}$, and initialize each worker with submatrix $U^g_{ij}$, starting iteration number $g$, needed maximum iteration number and the number of neighbors it has.
This will cause the computations to resume in the reconfigured system from the same point at which the old system was halted.

The reconfiguration problem may now be stated as follows:

**The Reconfiguration Problem**: The problem is to perform the tasks enumerated above with minimal collaboration from workers, with only the following requirement: each worker should reveal its current local iteration number and submatrix via its state port when requested to do so.

As we shall see below, POLYLITH requires considerable collaboration from workers in order to accomplish dynamic reconfiguration. Thus, unless workers had been designed and implemented *a priori* to provide for the expected collaboration, POLYLITH could not be used to accomplish the required reconfiguration task. On the contrary, the only collaboration required by RESTCLK is that each worker reveal its *local iteration number* and submatrix via its *state port* when requested to do so. Other than this, no participation is required from the workers. Thus, in RESTCLK it is not necessary that workers should be designed and implemented anticipating *a priori* the need for such dynamic reconfigurations.\(^1\) As far as we know, RESTCLK is the only paradigm that allows this kind of *transparent dynamic reconfiguration*.

### 9.1.2 RESTCLK Solution to the Reconfiguration Problem

**Worker Grid Network in RESTCLK**

The initial RESTCLK network for solving the Jacobi Relaxation problem is shown in Figure 9.6. The difference between the network shown in Figure 9.6 and the

\(^1\)The only claim we make is that worker codes would contain no special facilities for dynamic reconfiguration.
Figure 9.6: The Initial Configuration of Workers for Solving the Jacobi Relaxation Problem.
one shown in Figure 9.2 is in the structure of communication lines. Whereas Figure 9.2 uses passive communication lines between workers for exchanging boundary values, Figure 9.6 shows active RESTCLK pathways with clock rings, watch rings and agents. In each such pathway, a watch ring connects an obj-port to an agent on a clock ring. The other agent on the same clock ring is connected to another obj-port through a second watch ring. Arrows shown in the figure running parallel to these RESTCLK pathways indicate direction of transfer of boundary value data. Thus the top line on the right of $W_{22}$ that connects it to $W_{23}$ in the figure carries data from $W_{22}$ to $W_{23}$, and the pathway below it carries data the other way. For convenience Figure 9.7 shows an expanded view of these pathways connecting $W_{22}$ and $W_{23}$, shown with bold lines in Figure 9.6. $W_{22}$ will communicate with $W_{23}$ using the top pathway in this figure as follows:

Communication session will start when the agent $a_1$ in Figure 9.7 sends an *advance* to the port $p_1$ of $W_{22}$. At this point data in this agent’s private memory (not shown in the figure) will be empty. Thus port $p_1$ will not have any input data to read. During the active period of this communication session, when the boundary values to be sent to $W_{23}$ are ready for transmittal, $p_1$ will write them in its private memory (also not shown in the figure) and send a *release* and *trigger* to $a_1$. In the meantime $a_1$ will be waiting for the receipt of these signals.
When \texttt{a1} senses satisfaction of \textit{agreement protocol} it will copy data in the private memory of \texttt{p1} into its own private memory and forward it to the other agent \texttt{a3} on the same clock ring. When \texttt{a3} gets communication control it will send an \textit{advance} to the port \texttt{p3} of worker \texttt{W}_{23}. Receipt of this advance by \texttt{p3} will cause \texttt{W}_{23} to copy data from the private memory of \texttt{a3} into its own memory. After copying, \texttt{W}_{23} will cause its port \texttt{p3} to immediately \textit{release} and \textit{trigger} its agent \texttt{a3}. At this point the output data in \texttt{p3}'s private memory will be empty. Thus, \texttt{a3} will clear its private memory and transfer communication control back to \texttt{a1} and the whole cycle will repeat again. Of course, while \texttt{W}_{22} is thus transmitting boundary values to \texttt{W}_{23}, \texttt{W}_{23} may simultaneously be transmitting its boundary values to \texttt{W}_{22}. Thus asynchronous data exchanges over the entire net in the grid shown in Figure 9.6 may occur simultaneously.

In this scenario one may note that the active period of agent \texttt{a3}'s communication session with \texttt{p3} could be very short compared to the active period of agent \texttt{a1}'s communication session with \texttt{p1}. This is because \texttt{a1} will have to wait in each session for \texttt{W}_{22} to complete its computations, which will occur only after \texttt{W}_{22} has received all boundary values from all of its inputs. However, agent \texttt{a3} will sense agreement protocol satisfaction immediately after \texttt{W}_{23} had completed copying data from \texttt{a3}'s private memory. It does not have to wait.

Thus, communication control around the parent clock ring of \texttt{a1} will keep switching back and forth between \texttt{a1} and \texttt{a3}. As shown in Figure 9.6, depending on its location in the grid a worker may have anywhere between 2 to 4 neighbors. Thus in each cycle of computation a worker will receive 2 to 4 boundary values from its neighbors.

**Worker Algorithm**

The worker pseudo-code used in the RESTCLK solution of the Jacobi Relaxation problem is shown in Table 9.1. The code has an \texttt{INITIALIZATION} part followed
by an **iterations** part. In the initialization part worker receives its initial submatrix, its starting iteration number, needed maximum iteration number and the number of neighbors it has. It may be noted that all these values will be received by a worker *via* its state port. The **iterations** part consists of two phases: **communication** and **computation**. In the communication phase the worker exchanges boundary values with each of its neighbors and, if there is a request for the current submatrix or current local iteration number send them out immediately.

The **computation** phase will begin after a worker as received all boundary values needed for the current iteration, and has sent to its neighbors all the boundary values computed in its previous iteration. Notice, if any of the needed boundary values is not received by a worker, then worker will not be able to proceed to its computation phase. This will effectively halt all computations performed by the worker.

**Coordinator Algorithm**

When a need for dynamic reconfiguration is recognized RESTCLK will be asked to create and install a new object called **COORDINATOR**. This is shown in Figure 9.8. **COORDINATOR** will be connected to the worker grid network as shown in the figure. We will refer to watch rings that connect **COORDINATOR** obj-ports to pathways in the grid as **COORDINATOR probes**. For each worker the coordinator will probe one of its output pathways. Thus, as shown in Figure 9.8 for worker

\[ W_{ij} \text{ in row } i, \ 1 \leq i < 4, \ \text{column } j, \ 1 \leq j \leq 3, \]

the coordinator probes the output pathway going from \( W_{ij} \) to its neighbor \( W_{(i+1)j} \). These probes will be used to block the outputs being sent by \( W_{ij} \), respectively, to
Object WORKER
{
    /* INITIALIZATION */
    Wait to receive arguments needed for initial set up;
    Use the arguments to set up initial state consisting of the following:
    [sub-matrix, num_of_neighbors, starting_iter#, max_iter#];
    obj_port_list = construct a list of all obj-ports of this worker.
    
    /* ITERATIONS */
    for (iter = starting_iter#; iter <= max_iter#; iter++) {
        temp_obj_port_list = obj_port_list;
        
        /* COMMUNICATION PHASE */
        do {
            wait for advance along any obj-port in temp_obj_port_list;
            if (advance received on a "boundary-value recv" obj-port)
                read boundary values from bag,
                trigger and release the obj-port,
                and drop the obj-port from temp_obj_port_list;
            else if (advance received on a "boundary-value send" obj-port)
                output boundary value,
                trigger and release obj-port,
                and drop the obj-port from temp_obj_port_list;
            else if (advance received on the state-port)
                read request for state information (e.g., iter#, submatrix) from bag,
                output requested state information,
                then trigger and release state-port;
        }
        /* COMPUTATION PHASE */
        Relax();
        ComputeError();
    } while (temp_obj_port_list contains only the state-port);
}

Table 9.1: Pseudo-code for Worker Object.
Communication to retrieve Local Iteration #s of workers and submatrix values.

Figure 9.8: RESTCLK Network for Dynamic Repartitioning.
For example see probes in pathway connecting \( W_{22} \) and \( W_{42} \) in Figure 9.8 shown in bold lines. In addition to these, for each \( j \) \( 1 \leq j \leq 3 \), the output pathway from worker \( W_{2j} \) to worker \( W_{1j} \) is also probed. These probes will block inputs going into \( W_{1j} \) from \( W_{2j} \). See for example the probe attached to the input pathway going into \( W_{13} \) from \( W_{23} \) shown in bold line in Figure 9.8. Thus, for each worker in the grid one of its inputs and one of its outputs will be probed by the coordinator.

Let us consider the probe connection to pathway from \( W_{23} \) to \( W_{13} \) and analyze how the coordinator may use this probe to stop computations. As shown in the figure the coordinator probe is attached to the composite port of the first agent in this RESTCLK pathway, namely the agent that receives outputs from \( W_{23} \). We will refer to this agent as the probed agent of this pathway. This probe will be used by the coordinator to block data sent out by \( W_{23} \) from reaching \( W_{13} \), as discussed in Section 6.3.1 and recapitulated below:

There are two cases to consider. At the time the probe is introduced into the probed agent's composite port the probed agent (i) may be holding communication control over its parent clock ring or (ii) may not be holding communication control over its parent clock ring. In the second case communication control will reside with the other agent (the unprobed agent) on the same clock ring.

In case (i) the probed agent will be in its active state, waiting to receive outputs from its client obj-port, and as soon as probe connection is established it will send an \textit{advance} to the coordinator, as described in Chapter 4. In case (ii) no such \textit{advance} will be sent since the probed agent will be in its idle state. In this case the probed agent would have already forwarded output data it received from its client obj-port to the unprobed agent. After the unprobed agent had completed delivering this data to its client obj-port, eventually it will transfer clock ring communication control back to the probed agent. The probed agent will then send an \textit{advance} to the coordinator \textit{via} the probe.
Thus in either case the coordinator obj-port connected to the probed agent will receive an *advance* from that agent. Once this advance is received by the coordinator obj-port, the probed agent will be able to satisfy its agreement protocol only if both its client ports connected to it, namely the worker obj-port and the coordinator obj-port, send *release* and *trigger* signals necessary to satisfy the agreement protocol. Thus, the coordinator can prevent data flowing from $W_{23}$ to $W_{13}$ by simply not sending a *release*. This will cause $W_{13}$ to stop its computations at its local iteration number. At a later time the coordinator can cause $W_{13}$ to restart this halted computations and perform exactly one more iteration, by simply sending a *release* to the probed agent in the pathway. Thus the coordinator will be able to exercise step by step control over the iterative computations performed by worker $W_{13}$ after introduction of the probe.

Using all the probes shown in Figure 9.8 the coordinator can similarly exercise step by step control of computations performed by all the workers in the grid, once all probe connections have been established.

Until the current iteration numbers of all the workers become known, the coordinator carries out the following steps in a loop: Wait for advance on any of its probes; If the iteration number for the corresponding worker is already known, simply increment it and release the corresponding obj-port, otherwise, release the obj-port but only after querying the worker via its state-port for its iterations number and obtaining a response.

Finally when the iteration number for all the workers are known, the coordinator will set one plus the maximum of all these local iteration numbers to be the *global iteration number*, $g$. Thereafter, the coordinator will control step by step every worker, whose local iteration number is less than $g$, in order to make them go through requisite number of additional iterations so that they all arrive at the same global iteration number, $g$. It will thus bring the system to a consistent state.
At that point COORDINATOR will gather together from all workers, using its state probe pathway, their respective submatrices $S_{ij}^g$ and use these submatrices to construct the full matrix $S^g$. Thereafter COORDINATOR will destroy completely the worker grid and create the new $(m' \times n')$ grid. Each worker $W_{ij}$ in the new grid will then be initialized with its associated submatrix $U_{ij}^g$ obtained through repartitioning of $S^g$ into $(m' \times n')$ partitions, its new starting iteration number $g$, the needed maximum iteration number and the number of neighbors it has. This will complete the reconfiguration task.

In the next subsection we will compare this RESTCLK solution with the solution proposed in POLYLITH.

### 9.1.3 Comparison of RESTCLK and POLYLITH Solutions

**Analysis of RESTCLK solution**

In the reconfiguration process described above no worker participation was necessary other than communicating once their respective local iteration numbers and once their respective submatrices. Clearly, without these no reconfiguration would ever be possible. Interaction between COORDINATOR and workers, in a reconfiguration process may be depicted as shown in Figure 9.9. The top line
in the diagram is the coordinator time line and the bottom line is the time line of a worker $W_{ij}$. At time $t_0$, as shown in the figure, the coordinator requests the worker to send its local iteration number. Worker $W_{ij}$ shown in this diagram receives this query at time $t_1$ and responds immediately. The response from the worker is received by the coordinator at time $t_2$. By time point $t_3$ the coordinator will step the workers through their iterations to bring all of them to the global iteration number $g$. After this the coordinator queries the workers for their respective submatrices. It receives responses from all of them simultaneously at time $t_5$.

POLYLITH solution to this repartitioning problem also uses a coordinator. However, whereas in RESTCLK the coordinator is able to step workers through the needed number of additional iterations through external exercise of control with no collaboration from workers, the POLYLITH solution requires worker collaboration. Each worker object should have control mechanisms for this kind of stepping built into it a priori. Also, in each step each worker should communicate with the coordinator in order to find out whether additional stepping is needed. This scheme may be depicted as shown in Figure 9.10. The diagram in this figure shows the coordinator sending a ‘reconfig’ signal to each worker at time $t_1$. In response to this signal each worker will return the pair $(id, k)$, where $k$ is the

**Figure 9.10:** Illustrating Worker/Coordinator Interaction in POLYLITH.
local iteration number of the worker. After sending this output to the coordinator the worker should wait. Clearly workers will be able to receive and interpret this signal correctly only if they had been a priori programmed to do so. At the time the coordinator receives this information it may not know what the global iteration number should be. In this case, as shown in the figure the coordinator will send the signal, 'global iter# unknown,' to the worker. Worker will react to this signal by performing one more iteration, sending the next pair \((id, k+1)\) and again waiting for the coordinator's response, as shown in the figure. This process will repeat until the coordinator determines the value of the global iteration number. Once the coordinator finds the value of global iteration number, say \(g\), it will communicate this to the worker, who will then carry out enough iterations in order to reach the iteration number \(g\). After this the worker will send its submatrix to the coordinator. Clearly, each worker object should contain code needed for such collaboration. From this point on the tasks performed by the coordinator will be similar to the tasks performed by the RESTCLK coordinator.

The crucial differences between RESTCLK and POLYLITH solutions are (a) repeated communications between workers and coordinator are needed only in the POLYLITH solution (b) in POLYLITH reconfiguration will be possible only if worker object code had been augmented a priori to accommodate anticipated collaboration with the coordinator and (c) in POLYLITH waiting logic to synchronize coordinator/worker communications should be embedded inside each worker code. This leads to increased worker code complexity in POLYLITH: worker code has to satisfy not only matrix computation logic but also reconfiguration collaboration logic. In RESTCLK worker code has to satisfy only the matrix computation logic.

What is even more significant is that in RESTCLK the worker source code for a system with a capability for dynamic reconfiguration is identical to the worker source code without the reconfiguration capability. Thus one can dynamically
reconfigure a running system not designed \textit{a priori} for such reconfiguration without stopping the system and having to start all over again. \textsc{poly lith} shows how one could write a program with dynamic reconfiguration capabilities using the facilities provided by \textsc{poly lith}. On the contrary, \textsc{restclk} shows how a system not designed \textit{a priori} for dynamic reconfiguration, can be dynamically reconfigured in the middle of its operation with no required source code modifications. \textsc{restclk} is the only existing system we are aware of that can provide such capabilities.

\section{Dynamic System Evolution}

We discuss next the \textit{Evolving Philosophers Problem} presented by Kramer and Magee \cite{Kramer_1989}. We use this to illustrate how dynamic system evolution may be handled in the context of \textsc{restclk}, even if objects in the system had not been programmed in advance to accommodate such evolution. We begin with the problem statement in the next subsection.

\subsection{Problem Statement}

\textit{Evolving philosophers problem} is an extension of the \textit{Dining Philosophers Problem} first introduced by Dijkstra \cite{Dijkstra_1965}. In the dining philosophers problem a fixed number of philosophers are dining around a table with adjacent philosophers sharing a common fork. Every philosopher needs two forks in order to eat. No fork may be used simultaneously by two philosophers; this is called the \textit{safety} property.

At any point in time each philosopher can be in one of three states: \textit{Hungry}, \textit{Eating} or \textit{Thinking}. To move from hungry to eating state a philosopher should have two forks, which he may have to request from his left and right neighbors at the table (see figure). When a philosopher finishes eating he goes to his thinking
state. While thinking it is not necessary that a philosopher should relinquish his forks, unless his neighbors make a request for them. A philosopher may move from thinking to hungry state at any time.

The philosopher will continue thinking until he gets hungry. While in hungry state, if he does not have either or both the forks he will request for them as needed. This process will go on for ever with each philosopher around the table. The requirement is that no philosopher should remain hungry for ever; this is called the liveness property.

The problem is to design and implement a protocol for exchanging forks between adjacent philosophers which will guarantee safety and liveness. Thus, the protocol should make sure that every philosopher at the table will get opportunities to eat and no two will ever be using the same fork at the same time. Several solutions to this problem exist [8, 6].

The evolving philosophers problem extends the dining philosophers problem by allowing the number of philosophers around the table to change dynamically. A solution to this problem must ensure that dynamic changes in the number of philosophers can be carried out while preserving safety and liveness. Figure 9.11 shows an example of the evolving philosophers problem: an initial configuration with four philosophers dining around a round table evolves to one with five philosophers. Here addition of the new philosopher, $P_x$, to the table should be done while maintaining liveness and safety.

In this section we first describe a RESTCLK implementation of Chandy and Misra’s [6] hygienic solution to the dining philosophers problem. Then we show how RESTCLK facilities may be used for dynamic reconfiguration without requiring any change in the philosopher implementation. This may be viewed as an example of object source code reuse in RESTCLK, where the same object source code is used in different network contexts.

In Figure 9.11 the obj-ports used by the philosophers to communicate with
each other appear as the ears of the philosophers. For each philosopher his state-port appears as a knot of hair on top of his head; this is used to initialize philosophers at the time they join the table. The initial RESTCLK network for implementing the dining philosophers problem is shown in Figure 9.12. Here philosophers communicate with each other via RESTCLK pathways, each such pathway consisting of a clock ring, two agents and two watch rings.

9.2.2 Dining Philosophers Problem Solution in RESTCLK

We will describe below the salient aspects of the design that lead to the philosopher pseudo-code in the RESTCLK implementation, which is described later in this section.

- **Fork States:** A fork can be in one of two possible states: *clean* or *dirty*. A fork is said to be dirty if it has been used by a philosopher to eat. Always, philosophers will clean their forks before giving them to their neighbors. A clean fork will continue to remain clean until it has been used for eating.
Each philosopher will share his left fork with his left neighbor and his right fork with his right neighbor.

- **Requests for Forks**: With each fork we will associate a unique token. To assure safety we shall require that a philosopher may send a request for a fork to his neighbor only if he has in his possession the token associated with that fork.

- **Exchange of Tokens**: A philosopher will transfer the token of a fork in his possession to his neighbor (left token to the left neighbor and right token to the right neighbor) only when the fork goes from clean state to dirty.

Figure 9.13 illustrates how exchanges of requests, tokens and forks, satisfying the above rules, may occur between neighboring philosophers in our solution. The figure is self-explanatory.

The complete state of a philosopher may now be defined as follows:

\[
(\text{philosopher\_state}, \text{left\_hand\_state}, \text{right\_hand\_state}).
\]

The components of this triplet are,
1. **philosopher\_state**: As mentioned earlier, this can be one of eating, thinking or hungry.

2. **hand\_state**: The philosopher will have one hand state for each of his two hands. This can be one of the following: holding\_clean\_fork, holding\_dirty\_fork, waiting\_for\_token, can\_make\_request (does not have fork but holding token), waiting\_for\_fork.

We use these states in the pseudo-code shown in Table 9.2. The pseudo-code essentially describes the transitions and actions, some instances of which are shown in Figure 9.13.

### 9.2.3 RESTCLK Solution to Evolving Philosophers

We will use the very same code described above for each philosopher and initially set up the RESTCLK network shown in Figure 9.12. Here the objective is to introduce the new philosopher $P_x$ between $P_i$ and $P_j$. The bottom diagram in Figure 9.14 shows the final desired configuration together with the COORDINATOR
Object Philosopher
{
    /* INITIALIZATION */
    Wait to receive args needed for initialization via State Port;
    Use args to set up philosopher_state and left and right hand states;
    /* NORMAL PROCESSING */
    for (;;) {
        switch (philosopher_state) {
            case HUNGRY:
                while (have none or only one fork) {
                    if (hand_state(s) == CAN_MAKE_REQUEST)
                        Send out request(s) for fork(s) and trigger & release obj-port(s) and
                        Set those hand_state(s) = WAIT_FOR_FORK;
                    Wait for advance on obj-ports whose (hand_state != HOLDING_CLEAN_FORK);
                    if (data-bag contains fork or token)
                        Set that hand_state = HOLDING_CLEAN_FORK or CAN_MAKE_REQUEST;
                    else if (data-bag contains a request for a fork)
                        Send out fork after cleaning it and trigger & release obj-port and
                        Set corresponding hand_state = WAIT_FOR_TOKEN;
                }
                philosopher_state = EATING; eat_count_down = MAX_EAT_COUNT;
                break;
            case EATING:
                if (any hand_state == HOLDING_CLEAN_FORK) {
                    Send out token to that neighbor and trigger & release obj-port and
                    Set that hand_state = HOLDING_DIRTY_FORK;
                }
                while (eat_count_down--) eat();
                philosopher_state = THINKING; think_count_down = MAX_THINK_COUNT;
                break;
            case THINKING:
                while (think_count_down--) {
                    Look for advance on left & right obj-ports, do not wait;
                    if (any neighbor has sent request for fork)
                        Send out fork after cleaning it and trigger & release obj-port and
                        Set corresponding hand_state = WAIT_FOR_TOKEN;
                    else if (any neighbor has sent token)
                        Set corresponding hand_state = CAN_MAKE_REQUEST;
                        think();
                }
                break;
        }
    }
}

Table 9.2: Pseudo-Code for Philosopher.
used to arrive at that configuration. The top diagram shown the initial configuration in which the COORDINATOR sets up probes into both the agents in the pathway between \( P_i \) and \( P_j \). The steps taken by the coordinator for inserting the new philosopher is described in the next subsection.

**Coordinator Algorithm**

1. **Setup for Observation and Control**: This involves setting up the COORDINATOR and its associated probes to the dining philosophers, as shown in the top diagram of Figure 9.14. RESTCLK manager will create the
COORDINATOR in response to externally generated user stimulus.

The purpose of these probes is for the coordinator to find out which one of the agents $a_1$ and $a_2$ in the pathway between $P_i$ and $P_j$ (see Figure 9.14) is active and what is the most recent data that this active agent has delivered to its client obj-port. It may be noticed, COORDINATOR may get this data by simple accessing the active agent’s private memory without disturbing any of the philosophers. Thus no collaboration from the philosophers is needed. This data will be used by the COORDINATOR to determine what the initial state of the new philosopher, $P_x$, should be, as described next.

2. **Find Initial State of New Philosopher and Pending Activities**: Find the right hand state of $P_i$ and the left hand state of $P_j$ and use this information to derive the initial state of $P_x$ using Table 9.3. The first column of this table refers to the most recent data delivered by the active agent to its client obj-port. Without loss of generality, the active agent in the table has been assumed to be $a_1$ with obj-port $p_1$ as its client. The second and third columns refer to what the left and right hand states, respectively, of the new philosopher $P_x$ should be for data shown in the first column. The fourth column refers to what the new philosopher’s state should be for the same data. The last column specifies the activity that should be carried by the COORDINATOR at the end of the reconfiguration process, again for the data in the first column. The nature of this activity will be discussed later below.

3. **Create and Insert the New Philosopher between $P_i$ and $P_j$**: The manner in which the new philosopher is inserted around the table is shown in the bottom diagram in Figure 9.14: Connection from $a_2$ to $p_2$, shown in the top diagram of the figure is removed; a new connection is established between $a_2$ and $p_3$ as shown in the bottom diagram; new clock ring with
Data Delivered | (Left, Right) Hand States | Philosopher State | Pending Activity
---|---|---|---
REQUEST | (WAIT\_FOR\_FORK, HOLDING\_DIRTY\_FORK) | HUNGRY | send REQUEST
TOKEN | (HOLDING\_DIRTY\_FORK, WAIT\_FOR\_TOKEN) | HUNGRY | send TOKEN
FORK | (WAIT\_FOR\_TOKEN, WAIT\_FOR\_FORK) | HUNGRY | send FORK

Table 9.3: Table for State of New Philosopher & Pending COORDINATOR Activity.

primary agents $a_4$ and second agent $a_3$, shown in the bottom diagram of the figure, is created; and finally the obj-port $p_4$ is connected to the agent $a_3$.

It should be noted the connection (marked 3 in the bottom diagram) between $p_2$ and $a_4$ is not made at this time. This is carried out after the COORDINATOR has completed executing the pending activity shown in Table 9.3.

4. **Carry out Pending Activity**: At this point the COORDINATOR will first establish the connection, marked 1 in the figure, between one of its obj-ports and agent $a_4$. This agent will at this time be the active agent in its clock ring, since it was created as the primary agent of the clock ring. Thus, as soon as the COORDINATOR establishes the connection it will receive an advance from agent $a_4$. In response to this advance the coordinator will send to $a_4$ the data shown in the last column of Table 9.3.

5. **Make New Connections**: At this point COORDINATOR will shift the connection marked 1 in the bottom diagram to the connection marked 2 to agent $a_3$ in the same diagram. After establishing this probe to agent $a_3$ the COORDINATOR will wait to receive an advance from it, which will indicate that $a_3$ has become active. At this point the COORDINATOR will establish the connection between $a_4$ and $p_2$, marked 3 in the diagram.
6. **Send Initialization Data to** $P_x$: This consists of `LEFTHAND_STATE`, `RIGHTHAND_STATE` shown in the table above and the constants `MAX_EAT_COUNT`, `MAX_THINK_COUNT` shown in the philosopher pseudo-code in Table 9.2.

7. **Remove Coordinator**: In this step the `COORDINATOR` and all network components connecting it to the dining philosophers will be destroyed and removed.

This completes the dynamic addition of the new philosopher $P_x$ between $P_i$ and $P_j$. Dynamic removal of an existing philosopher will take place in a similar manner. We will not describe the details here. The important point to note here is that no collaboration was required from any of the philosophers in order to carry out any of the above described steps.

### 9.2.4 Comparison with Other Solutions

In CONIC [24] [25], in order to introduce the new philosopher $P_x$ between $P_i$ and $P_j$ all neighbors of $P_i$ and $P_j$ in the existing configuration should be *passivated* (see [25] for definition of passive state). Once a philosopher receives the *passivate* command from `COORDINATOR`, if it is currently in a non-HUNGRY state, or goes to a non-HUNGRY state, it will notify the `COORDINATOR` that it has become *passive*. After all the appropriate philosophers have been *passivated* the `COORDINATOR` will cause the new philosopher $P_x$ to be inserted between $P_i$ and $P_j$. This insertion process also will require considerable collaboration between $P_i$, $P_j$, $P_x$ and the `COORDINATOR`. After completion of this insertion process the `COORDINATOR` will *activate* all the previously *passivated* philosophers.

Since reconfiguration in CONIC requires collaboration from philosophers, the philosopher source code used in the *dining philosophers* problem cannot be used
directly for the *evolving philosophers* problem in CONIC. Code needed for collaboration has to be included in it. The resultant evolving philosopher code will be more complicated than the dining philosopher code.

Whereas CONIC *passivates* philosophers before insertion of a new one, POLYLITH *freezes* philosopher ports before insertion. However, since freezing is an external activity, POLYLITH requires less philosopher participation than CONIC. POLYLITH has to obtain the left and right hand states from $P_j$ and $P_i$ by requesting the philosophers to divulge them. This is the only collaboration required by POLYLITH. It may be noted, RESTCLK is able to infer these states from the observed message traffic, without having to freeze objects, because its access to messages satisfies the *immediacy* property mentioned earlier in Chapter 6. POLYLITH needs to resort to freezing and request objects to reveal their states because observation methods used in POLYLITH do not satisfy the *immediacy* property.

Clearly, *evolving philosopher code* in POLYLITH can be less complex than the code in CONIC; yet it is different from the *dining philosopher code*. Thus, in both CONIC and POLYLITH a Dining Philosopher game (DP-game) cannot evolve into an Evolving Philosopher game (EP-game) in the middle of an ongoing DP-game. To play the EP-game one has to stop playing the DP-game and start a new EP-game with the new code for evolving philosophers.

However, since the *evolving philosopher code* in RESTCLK is identical to the *dining philosopher code*, a DP-game in RESTCLK can truly evolve into a EP-game right in the middle of an ongoing DP-game. Thus RESTCLK allows true evolution of distributed software. Again, RESTCLK is the only system that can provide such capabilities.
9.3 Concluding Remarks

The central theme in the two examples discussed in this chapter has been that RESTCLK allows application programmers to develop simpler and more reusable source codes for application objects, and develop complex application systems which can truly evolve dynamically. As illustrated in the examples discussed here this is a natural consequence of *dynamicity and transparency* in RESTCLK.

All object codes illustrated in this chapter assume that objects have been designed to operate in the RESTCLK environment. This is not an unreasonable assumption.\(^2\) Thus *release, advance* and *trigger* control signals appear embedded in the pseudo codes displayed here. In each example we show objects used in reconfiguration environments are identical to those used in environments without reconfiguration. Thus object designs do not have to incorporate into them codes for reconfiguration.

It should be noted, however, that it is not necessary to program objects with RESTCLK control signals embedded in them. One could use any input/output statement to specify messages received and sent by an object. RESTCLK compiler may be written to translate these statements automatically to the requisite control signal emissions.

9.4 Other Related Work

Work on *dynamic reconfiguration* may be classified into three categories:

1. Problem identification and proposed solutions [24, 25, 15, 20, 21, 19, 44, 17, 18, 41, 34],

\(^2\) We show in Chapter 8 programming an object to function in a RESTCLK environment will in practice cost very little in terms of additional delays incurred in communication, when compared to any other agent mediated communication system. Thus there is little or no overhead to using RESTCLK.
2. Development of tools, languages and environments for dynamic reconfiguration programming [38, 5, 1, 26, 27, 23, 30, 45, 14], and

3. New paradigm that allows \textit{a priori} unplanned for dynamic reconfigurations. RESTCLK is the only work in this category.

Works in the first category above brought the problem of \textit{dynamic reconfiguration} into prominence and also proposed viable solutions to the problem which were widely accepted. We have already discussed details of the nature of the problem and proposed solutions. Some works focused mainly on a restricted version of this problem which involved only \textit{software replacement} [18, 40]. As is well known a central problem in dynamic reconfiguration is enforcement of application system consistency during and after the reconfiguration process. All the works in the first category discuss this problem.

Works in second category focused mainly on development of tools which made configuration and reconfiguration programming easier. Thus all of these works built on the basic solutions and methodologies proposed by works in the first category but did not by themselves introduce fundamentally new solutions or techniques.

All works in the first two categories considered only the problem of programming distributed system that could accommodate \textit{a priori} known patterns of dynamic reconfigurations during their lifetimes. All solutions and methods proposed in these works require some form of collaboration from application system objects in order to carry out dynamic reconfigurations. Thus some reconfiguration logic has to embedded into application object source codes.

Work on LGA (Law Governed Architecture) [34, 33] proposes doing dynamic reconfiguration by building reconfiguration primitives into the law that governs object interactions. However, reconfiguration examples for which use of LGA has been illustrated in [34], enforce only \textit{communication consistency}, namely making
sure that restrictions on objects on sending and receiving of messages are correctly enforced. These examples ignore the difficult problem of maintaining application system consistencies in distributed systems during reconfiguration tasks (such as those illustrated both in EP and dynamic repartitioning). It is not clear how controller in LGA can coordinate by itself dynamic reconfigurations if application system consistencies also need to be maintained.

This work also belongs to the first two categories mentioned above, in the sense reconfiguration requirements to enforce communication restrictions have to be \textit{a priori} planned for and implemented in the application system.

It may be noted that LGA requires involvement of relatively complex controllers in communication pathways. This is likely to introduce significant delays in communication. Careful analysis of overheads introduced by LGA is necessary before this architecture may be advocated for use in distributed systems. As indicated in [33], \textit{globality} is essential in LGA and hence LGA is less suitable when protocols to be represented as law can be localized to a few agents, that is, when heterogeneity rather than regularity is the norm.

We will conclude this chapter with a summary of the principal observations.

RESTCLK provides selective, localized and customized O&C. One may say, support for O&C provided by RESTCLK makes it possible to get reconfiguration capability for free in the following sense: At any point in the lifetime of an application system, programmers who have knowledge of application system objects behaviors, can formulate and encode needed \textit{reconfiguration coordinators} on the fly and incorporate them in the application system. These coordinators can then automatically carry out the necessary reconfiguration tasks while fully maintaining system consistency. Thus needed reconfiguration logic need not be anticipated and built into application system objects. There is, however, an assumption here: application system objects should have facilities to divulge information about their disclosure states.
Chapter 10
Conclusions

We have presented in this dissertation a new active, agent mediated communication paradigm for use in distributed systems, called RESTCLK. The RESTCLK communication paradigm assumes that objects will communicate through object ports to RESTCLK agents in a communication net. RESTCLK agents are responsible for gathering data from participating communicants, transforming them as necessary and forwarding them to other agents who in turn are responsible for delivering them to intended receivers. When needed, an agent may also temporarily withhold data from delivery. In a group to group communication all data written by members of a sending group will be delivered simultaneously to each member of the receiving group. Groups participating in group to group communication may themselves change dynamically in the middle of their communication sessions.

Coordination of changes is achieved in RESTCLK through Observation and Control (O&C) of data exchanged between objects in a system. RESTCLK O&C may typically involve interruption of data flowing among groups of objects in a system, testing and authenticating data in transit and/or modifying data while in transit. By temporarily stopping and holding the flow of data over selected parts of a RESTCLK network, or by modifying data while in transit, a RESTCLK agent can indirectly control the internal state transitions that may occur in objects in a distributed system that send and receive data. Systems using the RESTCLK services may employ management objects to direct and instruct RESTCLK agents on the kinds of O&C they should perform, when they should perform them and where. The management objects may also dynamically create and deploy (or
destroy and remove) RESTCLK agents and other network components over a
RESTCLK network (a) without interrupting on going activities in any of the
objects in a system and (b) without violating the consistency of a functioning
system, (c) while fully preserving the security and protection environment of
application systems.

The principal contribution that made RESTCLK feasible are the protocols
used by various ports in a RESTCLK network to communicate with each other.
We discussed in this dissertation,

- the structure of RESTCLK communication networks (Chapter 2);
- the protocols used by various RESTCLK ports and their important prop-
  erties (Chapters 3 and 4);
- significance to distributed systems design (Chapter 5);
- schemes for extending an application system's security environment to the
  RESTCLK network and the unique dynamic security and protection fea-
  tures that may be incorporated into a RESTCLK network to protect data
  in transit and monitor security enforcement (Chapter 7);
- features of the prototype RESTCLK implementation and related scalability
  and efficiency issues and possible extensions to RESTCLK (Chapter 8);
- details of two examples which were implemented in the prototype REST-
  CLK system to demonstrate how RESTCLK simplifies application object
  source codes, while at the same time enhances their reusability (Chapter 9);
  and finally
- comparative capabilities of RESTCLK with those of two other systems in
  its category, POLYLITH and CONIC, both of which proposed new and useful
  schemes for organizing communications in distributed systems (Chapter 9).
Some topics for future research & development were discussed in Chapter 8. We believe a pipelined organization of RESTCLK is essential for achieving the kinds of efficiencies needed for large scale applications. Also, the possibilities implementing RESTCLK within the RESTCLK paradigm, as also hardware supports for RESTCLK implementations, should be explored in detail. We believe, realization of these goals will have significant impact on distributed and communication systems technologies.

The important contribution here is that this work introduces a new programming paradigm. In conventional programming, mechanisms needed for setting up and managing communications are always viewed as being outside the scope of what is considered to be programmable. Thus conventional programming languages do not provide primitives for dynamically managing communication across networks. Abstractions needed for doing this did not even exist. In conventional programming, once data leave objects they are considered to be beyond programmatic control.

RESTCLK seeks to change this view: It calls for data to be always within the realm of programmatic control, whether they are inside objects or flowing between objects possibly across networks. The RESTCLK abstractions, such as clock rings, active agents, watch rings, the various kinds of ports and their associated APIs (Chapter 8), provide the basis for achieving such programmatic control. This work has shown how these abstractions can be effectively used in practice to implement such control, analyzed its important properties and demonstrated through examples many of the benefits that might be gained by using this new paradigm.

At this point the API presented here have the flavor of statements in an assembly language. Clearly environments that support higher level language constructs similar to those in [30] for creating and manipulating networks is needed. We expect such environments to evolve as more experience is gained in using this new
We have had several paradigmatic shifts in programming: from programming in binary bits, to assembly languages, to varieties of higher level languages, to Java with Java Virtual Machines. Every shift in paradigm was initially confronted with issues of loss of efficiency. In the long run, not only did the efficiency issues become moot, but they brought computers to where they are today in our society. This is not because they made programs more efficient, but because they made the task of programming much easier, programs portable and made it possible to develop very large software systems.

Dynamic management of large evolving software systems is now an important and critical problem area. New programming and organizational concepts are needed to address the problems. Indeed agent mediated communications [19, 16] were introduced to solve some of the associated problems in large distributed systems management. Use of RESTCLK paradigm can contribute towards solving many of the problems associated with dynamic management of large evolving software systems. We believe, as with higher level languages, acceptance of RESTCLK as a framework for large distributed systems development will come not because it made such systems any more efficient, but because it provided much needed capabilities for dynamic management of such systems during their lifetimes.
References


Vita

Souripriya Das

1964  Born in West Bengal, India.
1979  Secondary examination. Scottish Church Collegiate School, Calcutta.
1981-85  Attended Indian Institute of Technology (IIT), Kharagpur.
1985  B.Tech. in Computer Science, IIT, Kharagpur.
1985-87  Teaching Assistant, Dept of Computer Science, Vanderbilt University.
1988  M.S. in Computer Science, Vanderbilt University.
1987-90  Teaching Assistant, Dept of Computer Science, Rutgers University.
1990-94  Graduate Assistant, Dept of Electrical Engineering, Rutgers University.
1994  Senior Software Engineer, Digital Equipment Corporation.
1994-96  Senior Member of Technical Staff, Oracle Corporation.
1996-98  Principal Member of Technical Staff, Oracle Corporation.
1998-  Consultant Member of Technical Staff, Oracle Corporation.
1999  Ph.D. in Computer Science, Rutgers University.