A STUDY ON PROGRAM GRAPHS AND THEIR GENERATED MESSAGE FLOW

John R. Pickens

Notice: This Material may be protected by Copyright Law (Title 17 U.S.C.)

Technical Report # 67

May 1975

Department of Information and Computer Science

University of California, Irvine

This work was supported by a contract in Network Security from the Advanced Research Projects Agency.

John Pickens is currently attending UCI as a graduate student under intercampus exchange. His home department is the Department of Electrical Engineering and Computer Science at UCSB.

ACKNOWLEDGEMENTS

I wish to thank my doctoral committee co-chairmen Dr. Roger C. Wood (UCSB) and Professor David J. Farber (UCI) for their discussions on the work in this paper.

Special thanks is due Professor Farber for the countless discussions and special meetings arranged geographically equidistant between Santa Barbara and Irvine, and for his master plan of which this work is only a small part.

My deepest felt thanks is due to my wife Melinda and son Jeremy for their patience in enduring the ongoing trials of pursuing an advanced degree.

Introduction

The thesis of this working paper is that much information may be derived regarding distributed program behavior by observing and augmenting message flow. The progr~mming environment which is implied here is one in which modules with well defined boundaries, such as independent machines or independent address spaces, communicate with each other via a message oriented interprocess communications system.

Graph models may be used to represent distributed processes, e.g. see GOST, Chapter 3 for a machine distributed at a very primitive level. However, little has been done to examine the relationship of a distributed program's graph model to the message traffic it generates. Thus, this paper reports on the study of using the UCLA Graph Model of Computation to model a distributed process, and concentrates particularly on the observed message flow. The sections which follow explain the choice of the ARPA Initial Connection Protoccl (ICP) as the example to study, discusses the issues which surface because of this example, and suggests topics which might be candidates for future research. This paper assumes familiarity with the ARPANET Host-Host protocol (MCKENZIE), Initial Connection

Protocol (POSTEL1, POSTEL2), and the UCLA Graph Model of Computation (CERF, GOST, POSTEL2). One more comment, before proceeding, is that the primary contribution of this paper is seen as providing heuristics rather than formal algorithms with respect to monitoring distributed processes.

Some Goals of the Research Underlying this Paper

The research within this paper fits into a larger project which is investigating top down designs of distributed programming systems. A number of assumptions are made about program environments for this project; they are stated here in order to outline the context of this work.

A distributed program is a collection of independent, asynchronous, cooperating processes. Each process, or module, communicates with other processes only via message sending. The address spaces of the processes are independent and, in practice, may be affiliated with separate machines. Control flow is specified independent of module function, and in a way that may be used to build a program graph.

Given module, control flow, and intermodule

communications specifications, it is desired to investigate several issues related to the monitoring of such an environment.

First, little insight exists into the nature of the message traffic generated by actual program graphs. Thus a study of the relationship between program graphs and message traffic is a necessary precursor to any proposal for a monitor.

Second, it should be determined to what extent program graphs may be verified via an examination of message flow. E.G. Can all graph vertices be seen to initiate and/or terminate?

Finally, it should be determined just how such a programming environment might be monitored. Since it is desired to give the communications system enough information to monitor the status of program graphs, several approaches can be taken. First, investigate a communications structure in which no redundant control information is transmitted with each message. Second, investigate a communications structure which eases the task of the monitor by adding control information to each message. A tradeoff exists between monitor complexity and message complexity which should be studied.

Rationale for the Choice of Initial Connection Protocol

The approach of this work is to adopt an existing distributed process, ICP, and study its message and graph behavior. The results of this work may serve as input to future research. There are several motivations for investigating the Initial Connection Protocol.

First, the ICP process is _distributed_. Mutual action on the part of two processes is required for ICP to be successfully accomplished. In addition one may not be able to determine the complete state of a particular ICP only by looking at the state of one or the other of the two processes.

Second, the ICP process is a _parallel_ one. At its maximum, each party to an ICP may have 3 parallel requests outstanding (two connects for the send and receive connections and one close for the initial "contact" connection).

Third, the ICP is _common_ to, and is a _building block_ for, other more complicated distributed processes. For example, the File Transfer Protocol, the Remote Job Submission Protocol, and Telnet all depend upon ICP to initiate their sessions. Since the ICP is a building

block, it is hoped that by studying it one might see how low level processes may be represented on higher level program graphs.

Fourth, the ICP can be and already has been modeled with the UCLA Graph Model of Computation (POSTEL2), and thus part of the effort of generating a correct graph is alleviated.

Details of the ICP Program Graph

The ICP graph found in Jon Postel's Thesis (POSTEL2, pps. 104-105) has been adopted and simplified for the purposes of this study (Appendix I contains the resultant graph). Namely, the arcs modeling sockets as resources (U3,U2,U,S1,S0) and diagnostic arcs (NOTU,NOTL,NU3,NU2,NS0,NS1) have been removed. Also the vertices 18,19,20,21,22,23 have been removed. The graph therefore is much simplified and only contains data pertinent to our study of communications behavior. The arcs remaining represent message transmissions and sequencing rules.

One inadequacy which should be noted is that the program graph does not allow for asynchronous events. For example, it is perfectly valid for the SERVER to respond

to the USER's initial connect (CU2L) with a close, which should be interpreted as a refusal. Or a network error might occur at any time during the graph's execution which should be handled in a special way. These problems, though important, are seen a applying to a future study on using graph models in programming languages.

It may also be pointed out that while most arcs passing between the User and Server portions of the graph represent normal ARPANET messages, there is one exception and that is the arc labeled RFNM. RFNM represents an acknowledgement, which actually accompanies every message transmission. At least one reason for this arc is to prevent a race condition between the server's data (DATA) and the server's close (CLSL). If the close request passes the data in the network, then the data may be lost. In addition, RFNM represents a rèsponse from an NCP and does not imply termination of vertex 7. Stated differently, RFNM is a phenomenon associated with a lower level process, namely single message delivery, which is not explicitely modeled on this graph. If one specifies that a message on a given link must be acknowledged before another one may be transmitted, then RFNM may be eliminated. Notwithstanding such deficiencies, we will leave the graph as it is.

Communications Behavior of ICP Program Graph

Given the ICP program graph, let us now proceed to study the ICP communications behavior. We first note that what is desired is to identify how control flow information is imbedded within messages. We are trying to identify what information a communications monitor needs in order to follow the execution of the ICP program graph. We may speculate at the outset that there is probably a sequencing relationship between messages, although we don't know how to arrive at that relationship just yet. We may also say that messages will probably have to be decoded in order to identify their correspondence with token flow on the ICP graph. Otherwise, we make no apriori statements about the relationship between messages and the program graph.

The study proceeds as follows: First, each arc on the ICP program graph is enumerated, in detail, as to the message traffic it generates. Then a partial ordering graph is defined which delineates the sequencing of messages. With the enumerated message traffic, which allows us to correlate arcs and messages, and the partial ordering graph, which identifies the valid message sequencing, we have potentially enough information with

which to program a monitor for the ICP program graph. Thus the next step is to identify the control information dependencies, i.e. the information which must be extracted from each message in order to successfully identify other messages which appear later in time. When this is done for the ICP graph, it is shown that a conflict exists between the timing requirements enforced by control dependencies and the timing requirements allowed by the fully parallel partial ordering graph. Finally, we determine which vertices may be inferred as having executed by observing only the message flow.

Let us now enumerate the message traffic which corresponds to token movement on the ICP program graph. The table which follows details this message flow. To aid in understanding the table the following clarifications are needed. "U-->...-->S" and "U<--...<--S" indicate the direction of message flow. FROM and TO identify the nodes which lie at the head and tail respectively of each arc (see Appendix I). The last column, DATA, contains symbolic representations of commands and data sent over both the control-link and data links. RTS (connect from receiver to sender), STR (connect from sender to receiver), ALL (allocate buffer space for send), and CLS (close connection) all occur on the control link. LINK1,

LINK2, and LINK3 are the symbolic link numbers for the initial contact connection, User-->Server data connection, and User<--Server data connection respectively.

ARC	FROM	TO	MESSAGE
CU2L	2	4	U --> RTS (U , L , LINK1) --> S
CL2U	4	5	U <-- STR (L , U , SIZE1) <-- S
OPEN	5	6	U --> ALL (LINK1 , MSGS1 , BITS1) --> S
DATA	6	7	U <-- numeric value of S on LINK1 <-- S
RFNM	7	8	U --> rfnm --> S
CLSU	7	10	U --> CLS (U , L) --> S
CLSL	8	9	U <-- CLS (L , U) <-- S
CS1U2	12	13	U <-- STR (S+1 , U+2 , SIZE3) <-- S
CSU3	12	14	U --> RTS (S , U+3 , LINK2) --> S
CU2S1	11	16	U --> RTS (U+2 , S+1 , LINK3) --> S
CU3S	11	15	U <-- STR (U+3 , S , SIZE2) <-- S

Figure 1 -- Enumeration of ICP Graph Message Traffic

Note that eleven messages, counting RFNM, flow between user and server process. Note also that nothing is said about what happens after the ICP completes. I.E. The arcs which denote success of ICP may feed into another arbitrarily complex program graph.

We next apply to the messages a partial ordering on the time at which they each may be transmitted. The partial ordering we use here is a relationship between the messages such that any message's predecessor must occur prior to the message itself. The partial ordering is represented as a tree where any messages with a common

ancestor may occur in parallel. For the ICP graph the partial ordering is:

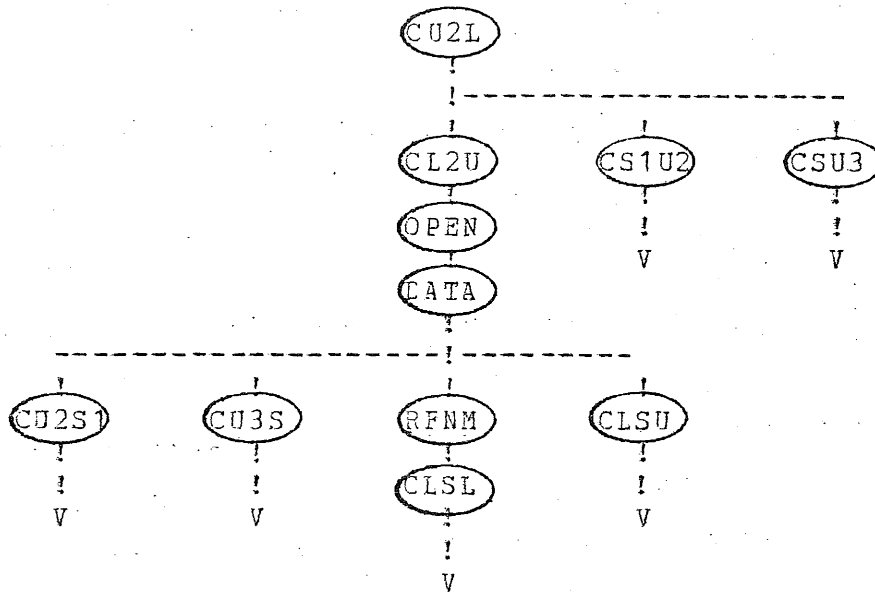


Figure 2 -- Partial Ordering of ICP Messages

Now that we know the message content and ordering for this particular graph, we may state the ancestral origins of each item of information that an intelligent monitor would need in order to follow the control flow of the ICP program graph. The total set of information required by the monitor is as follows: Socket numbers U, U+2, U+3, L, S, S+1; control link message types RTS, STR, CLS, ALL; and the link number, LINK1, for the initial contact connection. Additional information which may be determined, but is not used within the ICP program graph,

includes the link numbers of the final connections, and the byte sizes of all connections.

The socket numbers U and L must be provided, along with the program graph (or partial ordering), to the monitor explicitly. This is equivalent to saying that the user and server parties to an ICP must each inform the monitor of their intention to participate in the distributed process represented by the ICP program graph. The pair U,L initially uniquely identifies each instance of the ICP program graph for each pair of communicating processes. The rest of the information required to follow the execution of the program graph is derivable once the sockets U and L are known. The following figure delineates the information requirements of each arc on the ICP graph. The three columns contain for each arc the arc name, the message type, and the information extracted from the message representing this arc. The arcs are in four groups:

Arcs requiring U,L	Message Type	Can Extract
CU2L	RTS	LINK1
CL2U	STR	SIZE1
CL3U	CLS	----
CL3L	CLS	----
Arcs requiring LINK1		
OPEN	ALL	----
DATA	LINK1 data	Socket "S"
Arcs requiring U+2,S+1		
CU2S1	RTS	LINK3
CS1U2	STR	SIZE3
Arcs requiring U+3,S		
CU3S	STR	SIZE2
CSU3	RTS	LINK2

Figure 3 -- Information Content of Messages

To compare the timing requirements imposed by the flow of control information within messages let us do the following: Superimpose on the partial ordering graph an ancestral graph in which each node has an arc to a predecessor node in which its required set of information is defined. This is done in Figure 4:

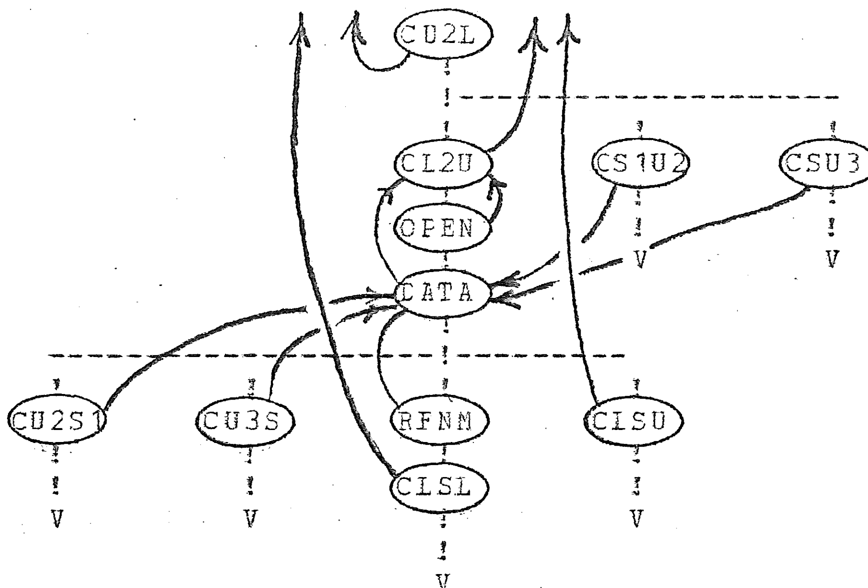


Figure 4 -- Ancestral/Partial Ordering Graph

Note that arcs CS1U2 and CSU3 may pass through the communications system prior to the socket number "S". ("S" and "S+1" are required by the last two groups of figure 3) Two solutions suggest themselves:

1. Supply "S" upon program graph initiation in the same way that "L" is supplied. This may be difficult if the allocation of "S" is done dynamically.

2. Ignore "S" and "S+1" completely. Upon close inspection it may be seen that these sockets are redundant information. "U+2" and "U+3" may not legally occur with any other combination than "S+1" and "S",

respectively. Thus, even though we sacrifice detail, it is possible to completely monitor, i.e. detect all state transitions within, the ICP program graph.

Now let us consider to what extent a given program graph may be verified via communications monitoring. In the case of this graph we infer that the passing of a message through the communications system (which represents the flow of a token on an arc) implies the termination of a vertex. In the ICP graph most vertices communicate directly with the network upon termination. In general graphs, fewer vertices may border upon the communications system, and more complicated mechanisms may be required to validate vertex termination.

Figure 5 shows the vertex terminations which may be implied as a function of message flow on the ICP graph. Note that vertices 13,14,15,16 may not be observed as having terminated by observing only message flow.

Vertex	Termination Implied by Arc
1,2	CU2L
3,4	CL2U
5	OPEN
6	DATA
7	RFNM,CLSU
8	CLSL
9,11	CU3S,CU2S1
10,12	CS1U2,CSU3
13,14,15,16	----

Figure 5 -- Implied Vertex Terminations

An Alternate Approach, Augmenting Messages

Up till now we have been investigating an unmodified protocol and asking the question, "assuming an infinitely intelligent monitor, may we accurately monitor the state of a program graph without introducing extra structure within messages?" We have determined, for the ICP graph, that this is possible. However, if this technique were extended to more complex environments, where an arc might represent an arbitrarily complex series of messages, it is possible that the overhead and necessary descriptive requirements might be expensive both in terms of execution and storage. Another approach is possible, which may simplify the operation of the monitor, and is sketched below.

In this alternate approach each message which moves into the communications system represents the traversal by

an arc of a token (at least in the ICP graph). Thus two items of information are required for the communications system to be able to monitor the flow. First is some sort of identification of the precise program graph to which this message pertains. For a common program graph like ICP there may be an arbitrary number of actual instances of the graph (somewhat analogous to reentrant code). Thus the identification should identify both the basis program graph and the particular instance of the program graph. Second is an indication of the particular arc on the graph to which the message pertains. These two items of information, if appended to the messages which flow through the communications system allow complete monitoring of the state of a graph without decoding the actual messages themselves. An approach like this has the obvious advantage of low execution overhead, but has the disadvantage that somewhere must be generated the correct identifying numbers for each graph and each arc. Just where such information is generated and how it is communicated to the monitor is not discussed in this study.

Overall Conclusions on Study of ICP Program Graph

This study of the ICP program graph's behavior as to

message communications has generated the following observations:

1. The UCLA Graph Model of Computation is a useful model for studying message behavior within the Initial Connection Protocol.

2. For those arcs which represent messages a partial ordering graph may be constructed which is more concise than the graph model and which preserves the potential for parallelism. The partial ordering graph contains only that structural information which is visible to the communications system and thus is potentially usable by the communications monitor to trace the flow of a program graph.

3. It can be useful to derive the ancestry of each bit of information that is required to associate a given message with its graph model and graph transition. When merged with the partial ordering graph this information makes it possible to identify when to augment messages in order to provide key information to the monitor at an earlier point in the graph's execution.

Future Research Directions

Several topics have surfaced as being logical successors to the research described in this paper. They are:

1. Continue the study of particular distributed processes which may be modeled by graph models. More specifically study FTP (File Transfer Protocol) and RJS (Remote Job Submission protocol). Both FTP and RJS depend upon ICP. RJS also uses FTP as a subroutine. The level at which arcs are traversed is known to be higher than that of simple message transfer (e.g. consider the token which passes from an RJS batch server to an FTP server when an output file is retrieved or delivered). Thus it is expected that some understanding will be obtained empirically of the macro properties of program graphs.

2. Investigate the use of the graph model as a programming language. More specifically study the modeling of asynchronous conditions and macro graph replacements. Investigate the graph/token duality (i.e. a simple token on one graph may be a complicated graph when viewed in finer detail).

3. Investigate the monitoring and debugging capabilities which normally might not exist without observing message traffic. Within a message oriented interprocess communication system which is able to follow the behavior of program graphs, it could be feasible to validate that a distributed process passes through only legal states, and that its behavior fits within prescribed bounds. Additionally, the setting of breakpoints and enabling of "ON" conditions could be made a function of states within the program graph. Without the program graph and its associated communications monitoring one would be restricted to monitoring only the local behavior of each individual process, and thus a more global view of the distributed program would be difficult to obtain.

4. Investigate the properties of the monitor itself. Should the monitor be built within or external to the communications net (with special filtering properties which allow monitoring information to be routed to it)? How intelligent should the monitor be and with what information should messages be augmented in order to most efficiently and completely monitor a distributed program? Should information supplied the monitor be static only or should procedural information, as might

be used to compute U+2 and U+3 from U, be allowed? What is the total set of services that the monitor might provide? What are the failsafe properties of the monitor? (e.g. monitoring its own program graph, resynchronization on interrupted monitoring)

5. It may be possible to develop a number of analytic and automated tools for monitoring program graphs. A few examples are:

   a. Derive the partial ordering graph from the program graph.

   b. Derive the program graph from the partial ordering graph.

   c. Derive the program graph by monitoring communications behavior.

   d. Determine sets of graphs isomorphic with respect to to communications.

   e. Derive canonical forms for program graphs according to given criteria (e.g. maximal parallelism vs. strict sequentiality).

REFERENCES

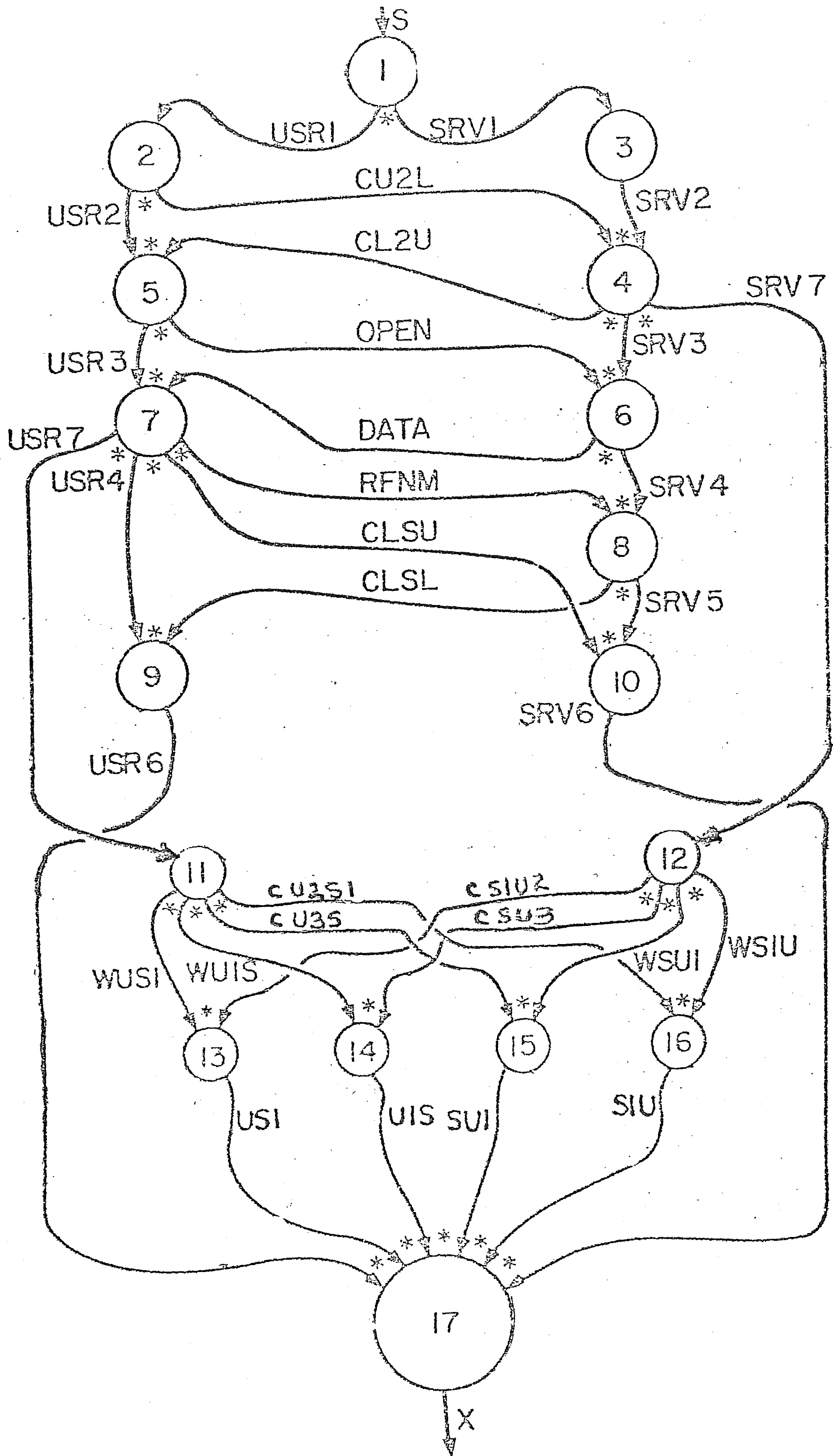
CERF Cerf, V.G. Multiprocessors, Semaphores, and a Graph Model of Computation, Ph.D. Dissertation, ENG-7223, Computer Science Department, University of California, Los Angeles, April 1972.

GOST Gostelow, K.P. Flow of Control, Resource Allocation, and the Proper Termination of Programs, Ph.D. Dissertation, ENG-7179, Computer Science Department, University of California, Los Angeles, December 1971

MCKENZIE McKenzie, Alex "Host/Host Protocol for the ARPA Network", ARPA NIC # 8248, January 1972

POSTEL1 Postel, Jon "Official Initial Connection Protocol", ARPA NIC # 7101, 11 June 1971

POSTEL2 Postel, J.B. A Graph Model Analysis of Computer Communications Protocols, Ph.D. Dissertation, ENG-7410, Computer Science Department, University of California, Los Angeles, January 1974


S
1
USR1
SRV1
2
3
CU2L
USR2
SRV2
CL2U
5
4
SRV7
OPEN
SRV3
USR3
7
6
USR7
DATA
USR4
RFNM
SRV4
CLSU
8
CLSL
SRV5
9
10
SRV6
USR6
11
12
CU2SI
CSIU2
CU3S
CSU3
WUSI
WUIS
WSUI
WSIU
13
14
15
16
USI
UIS
SUI
SIU
17
X