

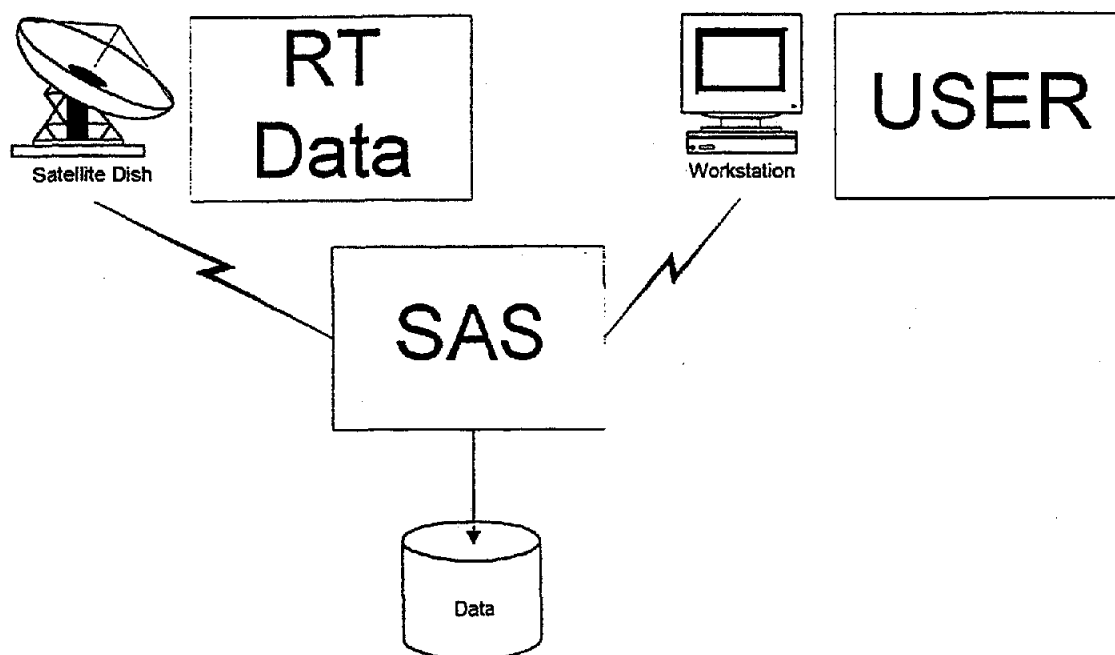
Using the SAS System for Real-Time Data Collection

Phil Busby, Live Data Systems

When real-time data is coming in, you have to be ready to handle it, or lose it. This tutorial covers several problems with programming real-time SAS data collection. Real-time data differs from standard data collection in that the processing of the data must be completed within a maximum known time interval. We don't know when the data will arrive, but when it does, our system must finish processing it in time to receive additional data that might arrive while we are processing the first batch.

One example of a real-time data processing application is the monitoring of vital signs of patients in a hospital. Another is radar scanner monitoring for the detection of incoming planes at an airport. These systems must respond promptly, or else they don't work. Typically they run in a multitasking environment where the data collection task is preemptive; that is, when data comes in, the task that receives it gets control and runs, while other tasks that were running on the system must wait. Since we don't know when data will arrive, then it follows that possibly there will be some tasks that never finish.

The SAS system is generally command-driven: you enter a command, and the system executes it and returns to you when it's done. Or you submit some SAS code; it runs, and you get your output. In the SAS world, the user's command has the highest priority. But in real-time data collection, the user's command input must not delay the data collection work, or data will be lost, so we have to adjust our view of how the SAS system must work to do real-time data collection. In the real-time data collection interface that SAS Institute has recently developed, the actual data collection task runs outside of the rest of the SAS system so that whatever commands you happen to enter do not delay the collection of incoming data.



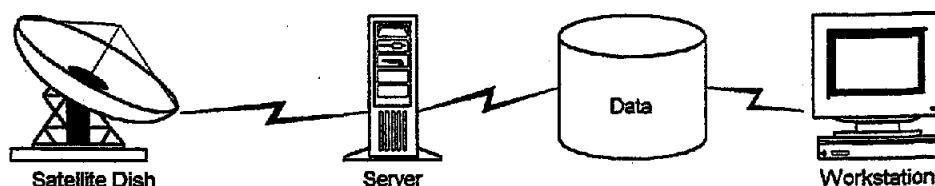
Our first application is illustrated by this diagram. The SAS/Trader system is used as an example of reading a broadcast: a single source of input is sending you data as it is generated, with a 511-character buffer that you cannot let overflow. Each time you read the buffer, you append it to your input stream, but how can you be sure some data was not lost?

The input data comes into the application via the computer's serial port, opened and read just like any external file, except that each time you do a read on the file, you specify a maximum buffer size for the input record, and the external file read function returns whatever there is-- usually somewhat less than your buffer size. So you ask for 512 bytes, and you might get back 100, or 2, or 64 bytes of data.

There is a special case you must consider when reading a broadcast: what if you do a read, and there is nothing coming in? That is, zero bytes in the buffer. If you have a SAS job doing nothing but data collection, then maybe it's OK for your task to block, or be suspended until there is some input data. If your task does other things, though, you may have to first ask the system how many bytes of data are available and skip the read if the answer is zero. In other words, do a non-blocking read if you want to continue your processing even if there's no input data.

The illustration shows a simple case of possible buffer chopping that the real-time programmer needs to watch out for. In the first chunk of data, we have a trade for IBM stock at a price of 66, followed by the start of another transaction where all we know is that it's for a stock whose symbol begins with the letter W. We can process the IBM trade and shift that data out of our buffer, then ask for more. The next chunk in the stream is appended to our buffer and we get a trade for WB at a price of 34, followed by the start of a transaction for IBM again. When the third chunk is brought in, we seem to have a completed trade for IBM at 65, but what if we waited too late to read the input stream, and the 5 was really a piece of the price for an entirely different stock? We actually have a syntactically correct transaction, but because the buffer was chopped off, our data is wrong.

In practice SAS/Trader can detect questionable trades and throw out prices that vary more than a certain percentage, and in addition there are verification checks put in by the quote vendor to correct erroneous data. This problem illustrates why you must have a CPU fast enough to read and process the data at the maximum baud rate you expect to handle.



***IBMt66*W / Bt34*IBMt6 / 5*Zt1**

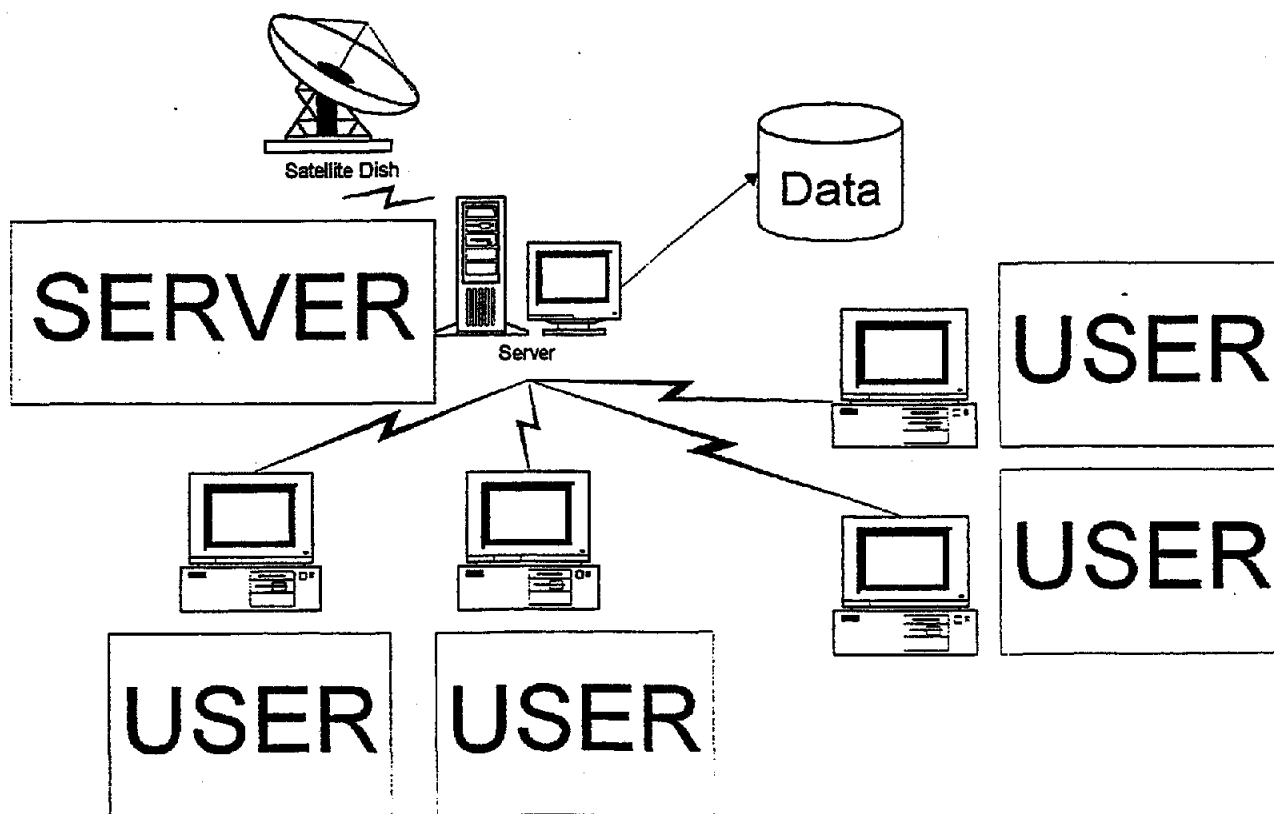
In this diagram, illustrated is the SAS/Trader server which collects market data (such as stock market prices) and supplies multiple clients with their custom portfolio of up-to-date information. Each client is a user on a separate workstation with his own custom portfolio. The client communicates to the server a request for "more data" and the server responds only when there is a trade or quote for a symbol in the client's portfolio. Since SAS/Trader runs under a single task, it is in a giant loop going around and around asking

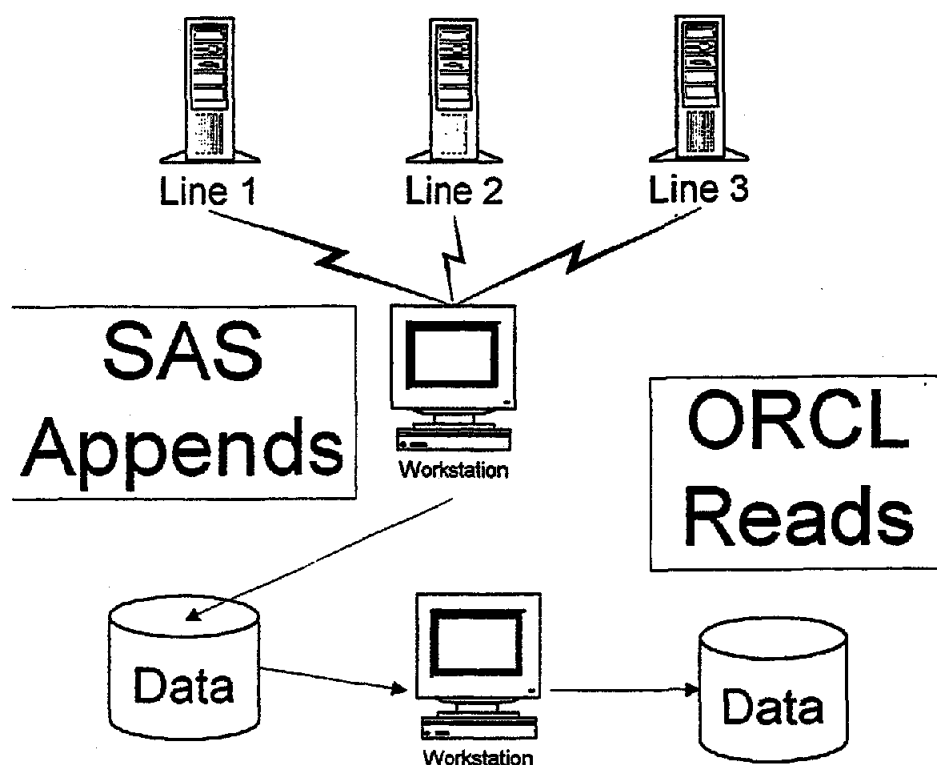
1. Is there more input data to process?
2. Is there a user command to execute?

Now, the answer to the first question is "Yes" for a client when the server sends the client a reply. As soon as the reply is received, however, and before the client does anything else, he asks the server for "more data please." In this way, the communication of requests for data and replies is overlapped with the actual processing of the replies when they are received. There is almost always a request for "more data" pending.

But what happens when the client goes off into the HELP or KEYS window, suspending his Trader task for 5 minutes? The server can't send data until the client asks for it—what to do? The server has collected some trades for a client who did not ask for more yet. More trades come in; the server has filled up his array of pending trades for the client, and now some trades are "lost."

Actually the server is writing a SAS data set of the transactions as they come in, but eventually the user gets done changing his KEYS and the SAS/Trader client task resumes. Now, when the client does ask for "more" after a 5-minute delay, the server does not have time to send him all of the trades that went by. To catch up, the server just tells the client, "you lost some trades while you were gone, but here's the current picture." At this point the server tells the client the latest information for every symbol in his portfolio, so at least he can see the last trade price for his stocks.

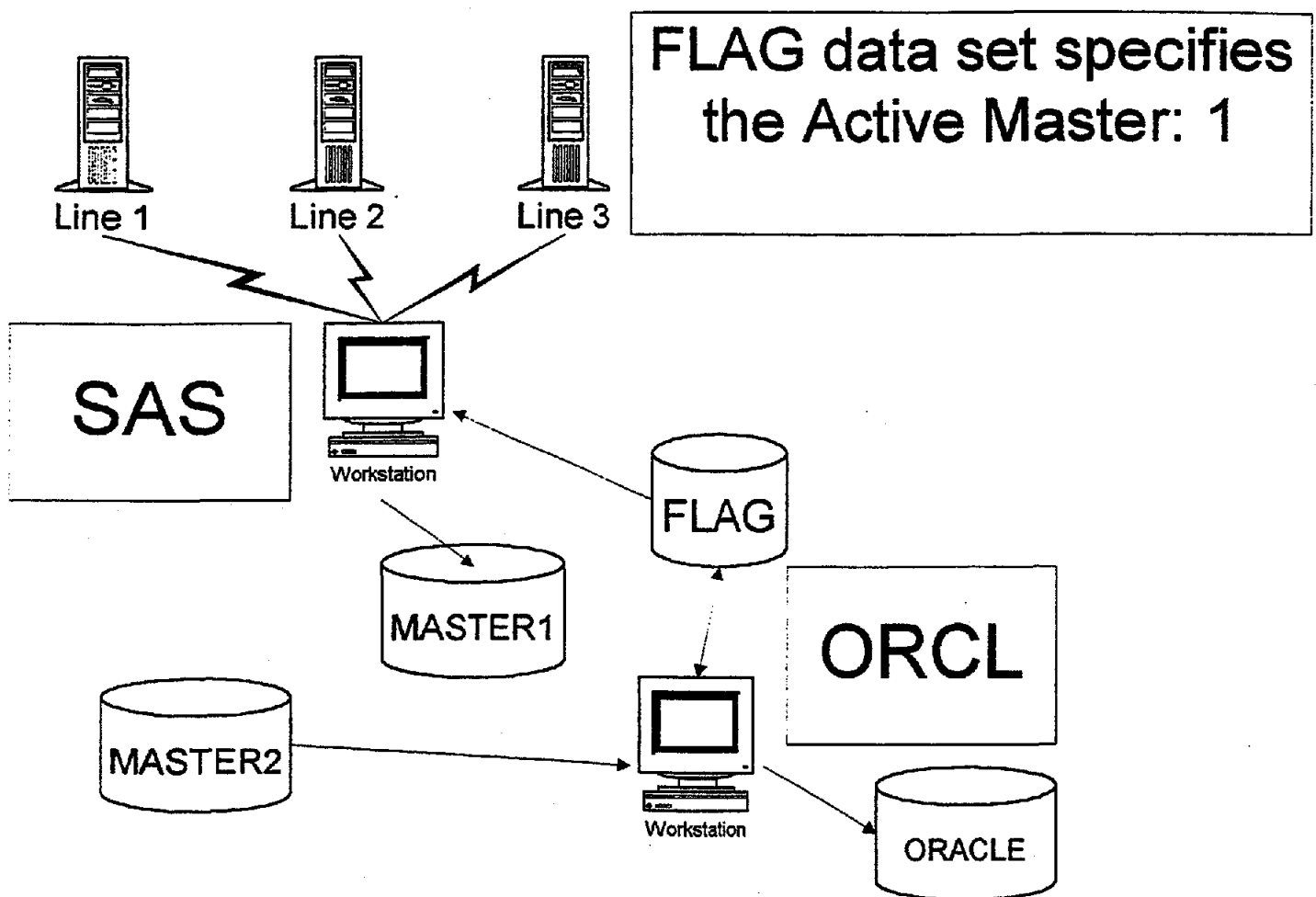


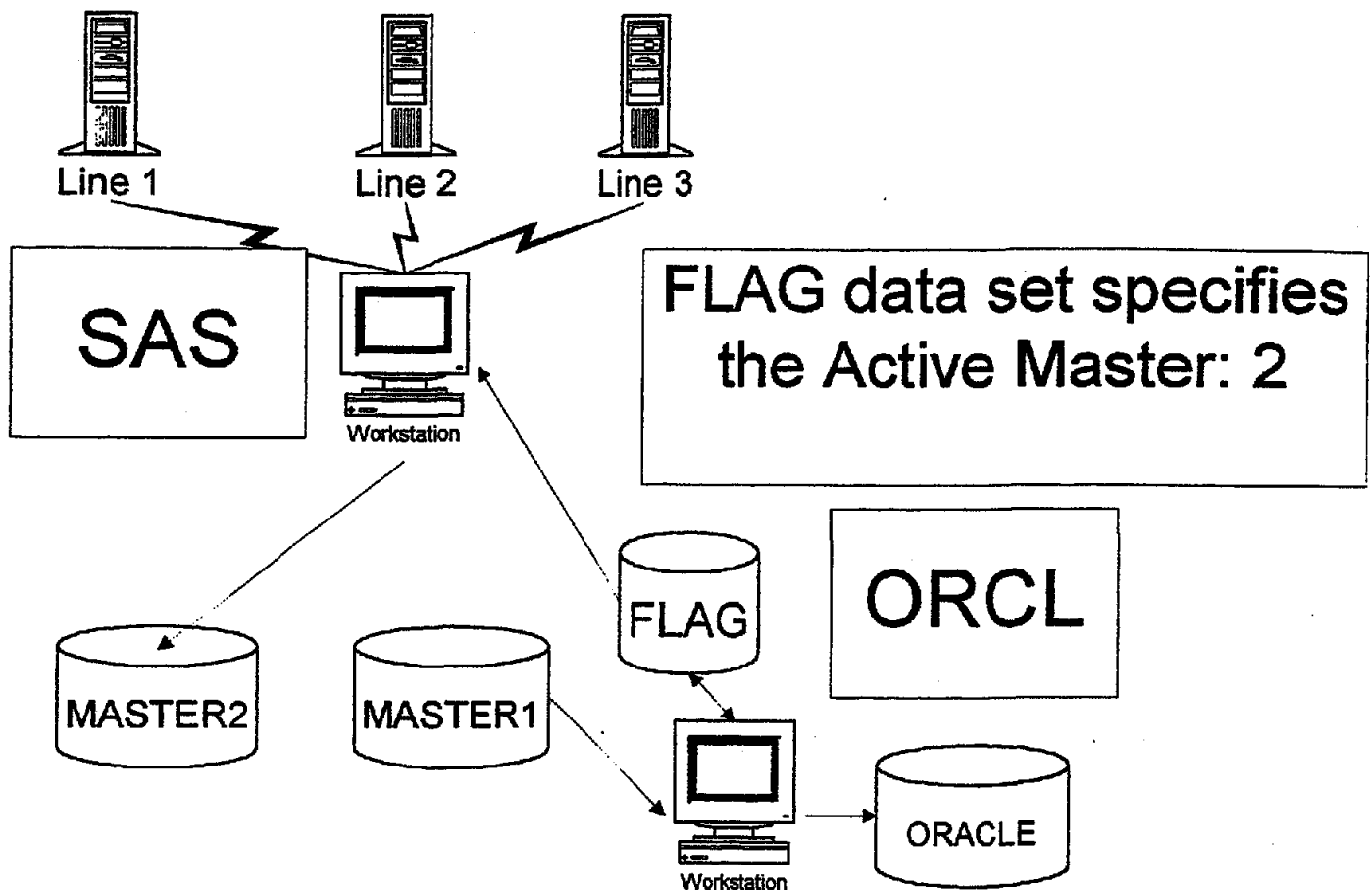


The second real-time example we have involves a manufacturing application: multiple ovens bake cookies. On each line, a measuring device records baking time and oven temperature; another measures how brown the cookies get coming out of the process. The devices send their data to SAS for analysis and archival. One SAS task writes observations to a SAS data set, but then it writes the same data to a separate data set for input into a separate SAS task for updating an Oracle database.

Now here's the problem. The task which is invoked to process all of the Oracle updates needs to open the SAS data set, read all of the data, reset the data set to empty, and close it. It runs once an hour and takes about a minute. But while the data set is being processed, the SAS real-time data collection task can't wait. It needs to write new observations to the data set during that minute that the Oracle processing occurs. Again, our system needs to respond with a maximum known delay, or data is lost, and cookies may burn up.

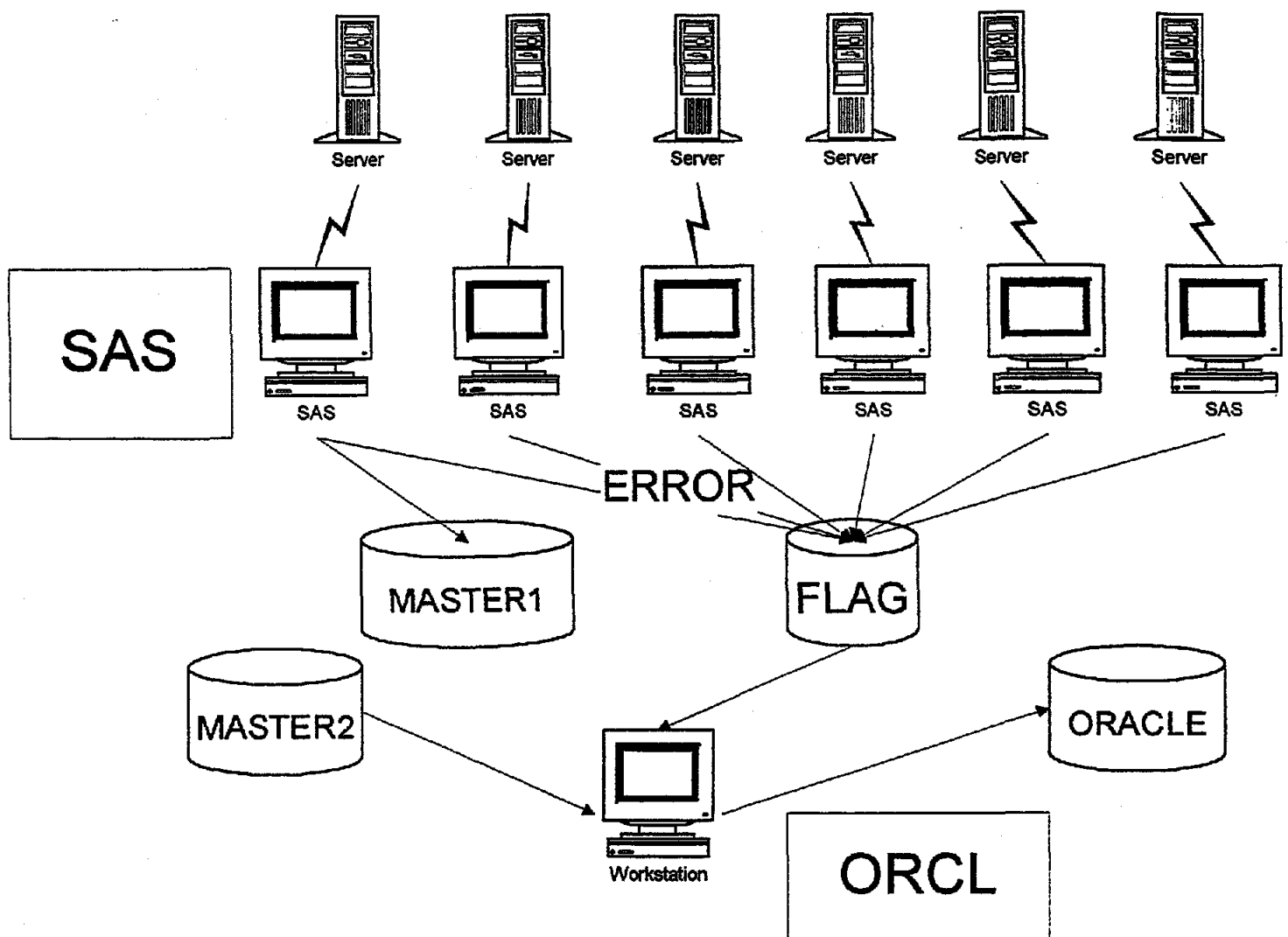
The next illustration shows one solution to the problem of simultaneous data set access. We create a Flag data set containing one observation and one 1-byte character variable called Flag. When the system is initialized, Flag has a value of '1'. The SAS real-time data collection task checks the value of Flag each time it gets an observation of cookie data to append. If Flag is '1', it appends to data set MASTER1; otherwise Flag is '2' and data set MASTER2 is the active master data set appended to. Once an hour, the SAS Oracle update task is invoked, and it changes the Flag value to '2', then processes and empties MASTER1 while MASTER2 is receiving the real-time data. An hour later, the SAS Oracle update task switches Flag back to '1' and processes MASTER2.





There's still one little problem with this system--what if the SAS data collection task reads a FLAG value of '1', but before it can append to MASTER1, the Oracle task changes FLAG to '2'? The SAS data collection task can lock the Flag data set until the cookie data set has been appended to, and if the Oracle task can't open the Flag data set, it can just go around in a loop a few times waiting until Flag is available.

That's fine with the Oracle task--it can wait a little while for the Flag data set, but now what happens when you put this system into production with six invocations of SAS all processing cookie data asynchronously, and they all just happen to try to lock the Flag data set at the same time? As the next picture shows, the second task to open the Flag data set for update will get back an error return code from the Open.



For the system to work, no task can lock the Flag data set. The solution here is for the cookie data collection tasks to open it for input only. When it comes time for the Oracle task to flip the Flag value, it opens, updates, and closes the Flag data set so that the cookie data collection tasks never have to wait more than a couple of tries to get the Flag value. Then the Oracle task goes to sleep for a while. During that time, all of the cookie data collection tasks have time to finish with the inactive master data set before the Oracle task processes it.

In summary, we have seen two real-time SAS data collection systems. When reading a real-time broadcast, you have to make sure you can handle the data as fast as it comes in. If you serve multiple client tasks, you have to handle the case where one of them crashes, and you must help them recover if they get suspended. When real-time data are handled by multiple computer systems, file contention can be a thorny problem. The real-time data input task has the highest priority. Other tasks, including the user interface task, may have to wait.