EXTENDED SHARED-VARIABLE SESSIONS

Karl Soop
IBM ESRI, 135 Chaussee de Bruxelles, B-1310 La Hulpe, Belgique

Roderic A. Davis II
IBM Data Systems Division, Box 390, Poughkeepsie, N.Y. 12602, USA

Abstract
This paper proposes two extensions to make shared variables of *APL* more useful: Shared variables that persist across *APL* sessions, and a facility to reject incoming offers.

1. Introduction

We observe that communication between two partners can span over several *APL* sessions, where each signoff and later signon of either or both partners forms an interruption that may be due to factors outside the communication protocol or the logic of the application. Using the shared variables of conventional *APL*, the partners must reinstate the communication where it left off, which, if at all possible, can be extremely cumbersome. This appears an unnecessary requirement, imposed by the shortcomings of the present mechanism.

The existence of a Shared-Variable Processor (SVP) with a memory for shared variables and ancillary information, suggests that this memory be made permanent on demand, thus allowing the communicating processors to bridge gaps in the on-going session. This forms the rationale for the first proposed extension (Section 2).

The second extension, Shared-Variable Rejection, partly stems from the first. It is motivated and presented in Section 3.

In the following, the term **specific,** with reference to a shared-variable attribute such as Access Control, refers to the value contributed by one of the partners, whereas **combined** refers to a value computed from the two specific values by the SVP.

We shall regard the (degree of) **coupling** as a boolean vector C made up of the partners' specific couplings, representing **connected** (1) and **disconnected** (0). The combined coupling is $+/C$, but is reported (e.g. by ⎕*SVO*) to the respective partners as $C\times+/C$.

We shall say that the SVP **discards** a shared variable V, when it deletes all information about it. V then ceases to be shared and neither partner can connect to it. Discard does not mean, of course, that the value of V need be lost; either partner can normally continue to use the variable in her own workspace.

2. Shared-Variable Persistency

To overcome the first deficiency discussed in the introduction, we propose **persistency,** a new attribute of a shared variable. This is a two-element boolean vector P, with a specific value associated with either partner. The value represents **persistent** (1), or **volatile** (0). The combined persistency is defined as $\vee/P$.

In current *APL*, only volatile shared variables are defined, which means that the SVP discards the variable when the coupling becomes 0 0. This does not happen with a persistent variable. Any one of the partners can later reconnect to the variable by making the appropriate offer. If the other partner matches the offer, communication can continue where it left off without any further preparation.

2.1 SPECIFICATION

Persistency is specified by a new system function, Shared-Variable Persistency. The expression:

$$W \; ⎕SVP \; N$$

sets the specific persistency of each shared variable in the character matrix N to corresponding values in the boolean vector W. Enquiry is effected by a monadic variant of ⎕*SVP*. The explicit result of both functions is a boolean vector, stating the **combined** persistencies of the variables in N, with zero wherever a shared variable is not specified. Thus (with the earlier notation):

$$C\times\vee/P$$

is reported to the respective partner.

Conformability and rules for the specification of N are the same as for ⎕*SVO*. The use of ⎕*SVP* does not engender an SVP signal.

2.2 EFFECT

We define the effect of persistence in terms of two partners A and B, who share a variable *V*:

1. When A retracts *V*, the SVP examines its attributes:

   * If A's specific persistency is zero, the SVP resets A's specific Access Control.

   * Then, if (and only if) the coupling becomes 0 0 and the persistency of *V* is 0 0, the SVP discards *V*.

   * Else, the SVP will retain all its information about *V* (namely the surrogate name, the value, the partners' processor Ids, the Access State, and all specific attributes).

2. A reconnects to the persistent *V* by offering it to the previous partner B. A must use the previous surrogate name, but the variable name need not be the same as before. If the coupling was 0 0, the offer is chronologically a new one.

3. On reconnection, A will see the variable automatically reinstated. This includes the value (if any), the Access State, and A's specific attributes.

4. A shared variable in a **general** offer may be made persistent, and will be retained on retraction as any other shared variable.

5. The co-domain of ⎕*SVQ* is expanded to report to A not only all outstanding offers to A, but also all persistent offers of coupling 0 0 where A is a partner. In short, A may use ⎕*SVQ* to find out which variables he may currently connect to.

Note, that to have the SVP retain the shared variable, it is enough if one partner makes it persistent. If A does but B doesn't, it is enough if, at a later stage, A alone makes the variable volatile to have it eventually discarded. If, on the other hand, both had made it persistent, both must make it volatile for it to be discarded. Note also, that the **coupling,** and the manner it changes, is not affected by the proposal.

3. Shared-Variable Rejection

In current *APL* the SVP retains outstanding offers to a processor A, and these are reported by ⎕*SVQ*, whether wanted or unwanted. The only way for A to "reject" an unwanted offer is to ignore it - that is, not to match it. Matching, and then retracting the variable, makes the SVP re-offer the variable to A.

An outstanding offer will persist as long as the shared variable persists: in current *APL*, until the partner retracts the variable. With the Persistency extension, the shared variable may persist indefinitely.

Now, if A intends to be responsive to other offers that it considers suitable, the continued presence of unwanted offers will require additional processing on its part. In order to get rid of them, A is dependent on the goodwill of his partners for retraction, possibly preceded by a persistency reset.

To avoid this problem, we propose that a processor be able to **reject** a shared variable. Rejection means that the SVP forces a retraction of the variable from each connected partner, and then discards the variable irrespective of persistency.

Rejection is achieved by a dyadic extension of ⎕*SVR*:

P ⎕*SVR N*

where *N* is a matrix of names, and *P* is a vector of corresponding processor IDs. Conformability of the arguments is the same as for ⎕*SVO*.

The effect of dyadic ⎕*SVR* is as follows:

1. If a row in *N* contains one name *S*, an unmatched offer of the surrogate name *S* to A from the corresponding processor in *P* is rejected. If several such offers are extant, the chronologically first one is rejected. The coupling before rejection may be 0 1 or (if persistent variable) 0 0.

2. A general offer can not be rejected.

3. If a row in *N* contains two names *V* and *S*, the shared variable specified by that variable name and surrogate (which may be identical) is rejected. The coupling before rejection may be 1 1 or 1 0; in the latter case, rejection is equivalent to retraction of a volatile variable by monadic ⎕*SVR*.

4. Rejection is always accompanied by an SVP signal to the partner.

The corresponding element in the explicit result of dyadic ⎕*SVR* is the combined coupling before rejection, i.e. the same as for monadic ⎕*SVR*.

One may object that the meaning of one versus two names in the argument is contrary to the semantics of the other system functions, such as ⎕*SVO*. But there is a precedent: The name matrix produced by ⎕*SVQ* consists of surrogates only. Note also the highly desirable construction *P* ⎕*SVR* ⎕*SVQ P*.

4. Application

Shared-Variable Persistency opens up several interesting applications, among others of the "mail-box" type. You can establish communication protocols with a number of partners, using shared variables with suitable surrogate names, and ascertain with suitable Access Control that your partner read your message before replying. Using the system function ⎕*SVS* of *APL*2 [Gerth], you may ascertain whether your partner has read your mail or not, even after he has signed off. This mode of operation caters to team work in a distributed environment, and is especially useful, e.g. in CAI applications.

You can make a one-sided connection by offering to any non-existent processor, thereby ensuring that the offer is never matched. If you make the variable persistent, it will be saved across workspace *LOAD*s and across sessions. You will be able to pass large objects from one workspace to another, reducing the transient workspace size in many applications. (Although not part of the current proposal, we suggest that a processor be allowed to offer a variable to itself for this purpose.)

Persistent variables have a key utility in connexion with another proposed extension: Processor Attachment [Soop], where one processor may sign on to another. Here the single user of a number of cooperating processors can close down the session in such a way that all shared variables, including their **states**, are conserved. She may then later resume the session exactly where she left off.

Rejection is useful in cases where the application provides some service to other applications. Typically, the multi-server processor must be responsive to new offers requesting service, while at the same time be able to "shut off" old requests that have been satisfied. The service function must also be able to reject offers that do not observe the proper protocol in terms of order, timing, and surrogate naming. The capabilities needed to achieve this are all the more important when the multi-server is **waiting** for shared-variable events (through ⎕*SVE* [Gerth]). Short of asking the potential partners to comply by doing their bit of clean-up, rejection will be the only method to **clear** undesired events so that waiting will not be vacuous. The proposed facilities are crucial when the waiting processor runs in an attached mode [Hartigan].

5. Practical Limitations

Although some SVPs use either partner's workspace to store the shared variables, many SVP implementations manage their own shared storage, and hence maintain a certain symmetry between the partners.

On an SVP crash, all variables in the shared storage are potentially lost. In particular, it may not be possible for either partner to recover the value of a shared variable. In addition, such SVPs usually apply a storage quota per user, and evoke an error to any user who attempts to share more than allotted to him.

The notion of persistent shared variables merely adds permanence to the shared-storage mechanism. This means that also a disconnected, but persistent, variable will use up quota. In the case only one partner has set the specific persistency to 1, the used-up quota may have to be transferred to him at the time of retraction. On a crash more information may get lost than without persistency.

Even with a quota system an SVP may get clogged with obsolete persistent variables that the partners may have forgotten about, despite the extension to ⎕*SVQ*. Therefore, the APL system engineer should have a utility to check, and eventually purge, the shared storage of obsolete material after an appropriate warning sequence.

References

John Gerth: "Toward Shared Variable Events, Implications of ⎕*SVE* in APL2", Proc APL83, APL Quote Quad 13,3 p.265, 1983

Bruce Hartigan: "AP19 -- a Shared Variable Terminal I/O Interface for APL Systems", Proc APL81, APL Quote Quad 12,1 p.137, 1981

Karl Soop: "Attached Processors in *APL*", Proc APL83, APL Quote Quad 13,3 p.129, 1983