Week 10 solutions

HOMEWORK
========

1. Send message to multiple clients.

We're supposed to do this all in the client program. Here's the code that sends messages now:

```
(define (im who message)
  ;;;Send message to who.
  (if (not
       (send-request (make-request whoiam who 'send-msg message) port-to-server))
      (close-connection)))
```

We want to change this so that WHO can be a list instead of just one name. The minimal solution is as follows:

```
(define (im who message)
  ;;;Send message to who.
  (IF (LIST? WHO)
      (FOR-EACH (LAMBDA (USER) (IM USER MESSAGE))
WHO)
      (if (not
           (send-request (make-request whoiam who 'send-msg message) port-to-server))
          (close-connection))))
```

If WHO is a list of names, instead of a single name, just call IM recursively for each name in the list.

This isn't perfect, because if something goes wrong, we keep trying to send even after closing the connection. There are many ways to fix that; maybe the easiest is to make sure we have a server before trying to send:

```
(define (im who message)
  ;;;Send message to who.
  (if (list? who)
      (for-each (lambda (user) (IF TO-SERVER (im user message)))
who)
      (if (not
           (send-request (make-request whoiam who 'send-msg message) port-to-server))
          (close-connection))))
```

2. Broadcast to all clients.

We're supposed to do the broadcasting in the server, so first we have to be
able to send a BROADCAST message from the client to the server:

```
(define (broadcast message)    ;; This is in im-client.scm
  (if (not (send-request (make-request whoiam 'server 'broadcast message) port-to-server))
      (close-connection)))
```

[Note that there is a procedure named SERVER-BROADCAST in im-server.scm that has nothing to do with this new feature!]

Then we have to modify the message-handling procedure in the server to
understand and process the BROADCAST message:

```
(define (setup-client-request-handler name port-from-client)
  ;;;Handle messages from the client.
  ;
  ;Only handles "send-msg" "BROADCAST" and "logout" messages.  *** FIXED
COMMENT!
  ;

  (define (client-request-handler)
    (let* ((port-from-client (socket-input client-sock))
   (port-to-client (socket-output client-sock))
   (req (get-request port-from-client)))
      (if (not req)
  (remove-client name)
  (begin
    (format logging "Received request: ~S~%" req)
    (cond
     ((equal? 'send-msg (request-action req))
      (let ((port-to-dst (find-port-to-client (request-dst req))))
(if port-to-dst
    (begin
      (format logging "Delivering message from ~A to ~A.~%"
      (request-src req)
      (request-dst req))
      (if (not
   (send-request (make-request (request-src req)
       (request-dst req)
       'receive-msg
       (request-data req))
port-to-dst))
  (remove-client (request-dst req))))
    (begin
      (format logging "User not found: ~A. Letting sender know.~%"
      (request-dst req))
      (if (not
   (send-request (make-request 'server
       name
       'receive-msg
       (format #f "User not found: ~A"
       (request-dst req)))
port-to-client))
  (remove-client name)) )))) )

     ((EQUAL? 'BROADCAST (REQUEST-ACTION REQ))
      (FOR-EACH (LAMBDA (CLIENT)
(IF (NOT
     (SEND-REQUEST (MAKE-REQUEST (REQUEST-SRC REQ)
CLIENT
'RECEIVE-REQ
(REQUEST-DATA REQ))
   (FIND-PORT-TO-CLIENT CLIENT)))
    (REMOVE-CLIENT CLIENT)))
(GET-CLIENTS-LIST)))

     ((equal? 'logout (request-action req))
      (remove-client name))

     (else
      (format logging "Unrecognized action requested: ~A. Letting sender know.~%" (request-
action req))
      (if (not
   (send-request (make-request 'server
       (request-dst req)
       'receive-msg
       (format #f "Unrecognized action: ~A"
       (request-action req)))
port-to-client))
  (remove-client name)) ))))
    ;; if other data ready, handle them now
    (if (and (not (port-closed? port-from-client))
     (char-ready? port-from-client))
(client-request-handler)) ))

  ;; Set up the handler
  (when-port-readable (socket-input client-sock) client-request-handler))
```

I wrote the new part (capital letters) by copying part of the cond clause for send-msg and modifying it as follows:

1. Change "send-msg" to "broadcast" (the name of the message).
2. Wrap a FOR-EACH around the call to send-request, so that we send a "receive-msg" to every client instead of just one.
3. Change the (request-dst req) to CLIENT (the formal parameter of the lambda expression) because the sender didn't specify a

particular recipient.

3. Discuss client-side vs. server-side processing.

Either task could be done either way.

The main virtue of doing the work in the client (as you can see by comparing the two solutions above) is that it's less work for the programmer, often, because only one program has to be changed. This also makes it less likely that the change will introduce bugs, because you don't have two different programs that have to agree. (This is even more true in the real world, where you're likely to have some people running the new version and other people still running the old version of the client software. If the new client is designed so that no server changes are needed, both new and old clients should still run without problems. When a change is made to the server, you have to take the trouble to ensure that the new server software still supports old versions of the client software.)

The main virtue of doing the work in the server is that it reduces the load on the network, at least in these examples. Sending a message to N people entirely through client-side changes means that the client software sends N separate requests to the server, which in turn sends N messages to the other clients. Doing the same thing by teaching the server software a new capability means that only one message goes from the originating client to the server, so the total number of network messages is N+1 instead of 2N.

4. Refusing messages.

This will be easier if we do it entirely in the client software, not only for the usual reason of keeping the changes local to one program, but also because each client can maintain its own list of refusees. If we do it in the server, then the server needs a global database of who dislikes whom. (In a real-world IM system, the server already has a database of users, because you have to enroll in order to have a username, so adding another item to the per-user database isn't so bad. But in our version, the server program has no permanent database at all.)

What about network efficiency? A refused message takes four network requests: sender to server, server to recipient, recipient sends refusal notification to server, server to sender. If we kept the enemies list in the server software, only two network requests would be needed. But we can hope that refusals aren't that common, so this won't be too much of a burden.

But this decision has a bad consequence, noted below!

Okay, our first task will be to develop a user interface to allow a user to

say whose messages should be refused. I'll invent two new user procedures:
(refuse-from "fred")
(accept-from "fred")
We'll have a global variable ENEMIES that contains a list of usernames:

```
(define enemies '())

(define (refuse-from whom)
  (if (not (member whom enemies))
      (set! enemies (cons whom enemies))))

(define (accept-from whom)
  (set! enemies (filter (lambda (name) (not (equal? name whom)))
enemies)))
```

Now we can modify the handling of an incoming message to see if we want
to show it to our user:

```
(define (received-msg from-whom msg)
  ;;;Handles message received from other clients.
  ;Change to GUI in future.
  (IF (MEMBER FROM-WHOM ENEMIES)
      (IM FROM-WHOM "YOUR MESSAGE WAS REFUSED.")
      (display (format #f "~%Message from ~A:~%    ~A~%~%" from-whom msg))))
```

That's about it! No other parts of the program have to be changed. But
there's one severe bug in the implementation so far: Suppose I'm on your
enemies list, and you're on mine. (It's not so unlikely for this symmetry
to occur, after all.) If I send you a message, your software will send me
back a refusal notice. Since that notice looks like a message from you,
I'll refuse it, and so I'll send you a refusal notice. These notices will
go back and forth forever! That really bogs down the network for everyone.

Here's a case in which using the server software to do some of the work
would have helped. We could mark the refusal notice as coming from "server"
rather than from the enemy, so the refusal notice wouldn't be refused.
Instead, we'll have to put up with not seeing the refusal notification in
these situations, but at least we can avoid the infinite message loop:

```
(define (received-msg from-whom msg)
  ;;;Handles message received from other clients.
  ;Change to GUI in future.
  (if (member from-whom enemies)
      (IF (NOT (EQUAL? MSG "YOUR MESSAGE WAS REFUSED."))
  (im from-whom "your message was refused."))
      (display (format #f "~%Message from ~A:~%    ~A~%~%" from-whom msg))))
```

[Note: I'm putting the changes in CAPITAL LETTERS to make them easy to find, but in fact two strings aren't equal if one is in capitals and the other in lower case letters. You'd want to be consistent.]

Another solution, requiring code in the server as well, would be to invent a new command SEND-REFUSAL that's like SEND-MESSAGE but specially marked so that the target client gets a RECEIVE-REFUSAL command, which it would know not to refuse.

5. Why the 3-way handshake?

In other words, why isn't it good enough to send the connection request from client to server, and an acknowledgement from server to client?

We want to make sure that we've actually successfully set up a two-way interconnection between the two ends. What if the server can get messages from the client, but the client never sees messages from the server? We want to find out about that, and notify the user that the connection wasn't set up successfully. The network software will "time out" if a reply doesn't come after a certain number of seconds, so both client and server can be confident that each can talk to the other.

(Of course it's still possible for a connection to be lost after some of these handshaking messages go by. In really critical situations, hosts will send each other "are-you-alive?" messages every so often, just to be sure that hasn't happened.)

This is the sort of thing you don't have to worry about when you're not doing network programming. Failures on a single computer are generally all-or-nothing; if anything works, everything works.

3.38 (a).

Peter then Paul then Mary: 100 -> 110 -> 90 -> 45.
Peter then Mary then Paul: 100 -> 110 -> 55 -> 35.
Paul then Peter then Mary: 100 -> 80 -> 90 -> 45.
Paul then Mary then Peter: 100 -> 80 -> 40 -> 50.
Mary then Peter then Paul: 100 -> 50 -> 60 -> 40.
Mary then Paul then Peter: 100 -> 50 -> 30 -> 40.

So the possible final values are 35, 40, 45, and 50.

(b). There are lots of extra possibilities. It's especially bad because Mary's command is

(set! balance (- balance (/ balance 2)))
which examines the value of balance twice, instead of the mathematically equivalent
(set! balance (/ balance 2))
which only examines the value once.  Here are just a few possibilities:

Peter examines balance, then Paul and Mary run in either order, then Peter sets balance.  Paul's and Mary's activities become irrelevant; the final balance will be 110.

Mary looks at balance, then Paul runs, then Mary looks at balance again to compute (/ balance 2), then Peter runs, then Mary sets the balance.  Peter's work is irrelevant, but Paul's isn't; Mary computes (- 100 (/ 80 2)) and the final balance is 60.

Paul examines balance, then Peter and Mary run in any order, then Paul sets balance.  The result is 80.

3.39 buggy serialization

Because multiplying x by itself is serialized, we can eliminate the case in which (* x x) gets two different values of x, which is the one that leads to a final answer of 110.

However, because the setting of x in P1 is not serialized along with P1's reading of x, we can still get the incorrect answer 100.

And because only the computation part of P1 is serialized, the incorrect answer 11 is also still possible.  It can't happen the way it's described in the book:

P2 accesses x, then P1 sets x to 100, then P2 sets x

but it can happen in this more complicated way:

P1 reads x and computes the value 100, then
P2 accesses x (still 10), then
P1 sets x to 100 (since this part isn't serialized), then
P2 sets x to 11.

(And of course the correct answers 101 and 121 are still possible.)

3.40

The smallest possible result is the one you get if P1 reads and squares 10, then P2 runs completely, and then P1 sets x to 100.

The largest possible result is the correct one, in which the
two processes run serially (in either order); the result is 1,000,000.

Any power of 10 between these limits is also possible:

1000: P2 cubes 10, then P1 runs completely, then P2 sets x to 1000.

10000: P2 reads x=10 twice, then P1 runs completely, setting x to 100,
then P2 reads x=100 for its third factor and sets x to 10*10*100.

100000: P2 reads x=10 once, then P1 runs, then P2 reads x=100 twice.

If the processes are serialized, only the correct result 1,000,000
is possible. Either P1 sets x to 100 and then P2 cubes that, or
else P2 sets x to 1000 and P1 squares that.

3.41 serialize reading balance?

The reason we need serialization in the first place is that if a
process makes more than one reference to the same variable, we want
to be sure its value is consistent throughout that process. But
reading the balance only looks at the balance once; how could
another process interrupt that?

Actually, this answer is a slight handwave. In order to be confident
about it, you must know that a number can be read from the computer's
memory in a single, indivisible hardware operation. This is true for
every modern computer, provided that the number fits in the usual
hardware representation of numbers. However, Scheme supports
"bignums", which are unlimited-precision integers. That's why in
Scheme you can compute the factorial of 1000 and get a precise
answer, even though that answer has 2568 decimal digits. Reading
a bignum requires more than one memory reference, and so Ben is
right if bignums are allowed.

The usual indivisible-lookup kind of integer can support values
up to about 2 billion, so this means that the bank that handles
Bill Gates' account should take Ben's advice, but the rest of us
don't have to worry about it.

3.42 when to serialize procedures

Ben is right. The key point to understand is that there are
three distinct steps in using a serializer:
1. Create the serializer.

2. Create a serialized procedure.
3. Invoke the serialized procedure.

Only step 3 involves concurrency problems. Ben's proposal is to reduce the number of times step 2 is taken, which is fine.

## 3.44 Transferring money

Louis is wrong, as usual.

In the case of transferring a fixed (given as argument) amount of money, there really are two entirely separate steps: Withdraw the money from one account, and deposit the money in another. Each step is serialized by the account itself. It doesn't matter if something else happens between the two steps! You might think "another process could change the value of the variable AMOUNT" but that's not so; AMOUNT is a *local* variable within this transfer, not shared with any other process. We only have to worry about synchronization when a state variable is shared among different processes.

Also, in the case of TRANSFER, the only constraint on correct answers when several transfers are done in parallel is that the sum of the balances should be the same after the transfers as before.

EXCHANGE has a more stringent requirement; after several parallel exchanges, the balances of all accounts should be the same as before, except in a different order. It's not good enough for the sum of the balances to be preserved; the individual balances must be, too. Since the amount to be transferred is a function of the starting balances, we can't let another process intervene between computing that amount and adjusting the balances.

## 3.46 Why test-and-set! must be atomic

Suppose the mutex is initially false.

P1 starts doing (if (car cell) ...) and gets the value FALSE, so it thinks the mutex is free.

Now P2 runs, before P1 does anything else. It, too, does (if (car cell) ...), reads the value FALSE, and thinks the mutex is free.

Then they both set the value to TRUE and claim the mutex.

We have to make sure that there is no intervening process between reading (car cell) and (set-car! cell true).

3.48 deadlock

The problem we're solving is that P1 and P2 want the same two serializers S1 and S2. Deadlock will happen under the following order of events:

1. P1 gets S1.
2. P2 gets S2.
3. P1 wants S2, can't have it, waits.
4. P2 wants S1, can't have it, waits.

Both processes wait forever because they're trying to use a busy serializer.

This problem appeared because P1 and P2 asked for the serializers in reverse order from each other. If they both use the same order, the worst sequence of events will be this:

1. P1 gets S1.
2. P2 wants S1, can't have it, waits.
3. P1 gets S2.
4. P1 finishes, releases both serializers.
5. P2 finally gets S1.
6. P2 gets S2.
7. P2 finishes, releases both serializers.

Here's the code:

```
(define make-account-and-serializer
  (LET ((NEXT-ID-NUMBER 1))
    (lambda (balance)
      (define (withdraw amount) ...)
      (define (deposit amount) ...)
      (let ((balance-serializer (make-serializer))
    (ID-NUMBER NEXT-ID-NUMBER))
(define (dispatch m)
  (cond ((eq? m 'withdraw) withdraw)
((eq? m 'deposit) deposit)
((eq? m 'balance) balance)
((eq? m 'serializer) balance-serializer)
((EQ? M 'ID) ID-NUMBER)
(else (error ...))))
(SET! NEXT-ID-NUMBER (+ NEXT-ID-NUMBER 1))
dispatch))))

(define (serialized-exchange account1 account2)
  (IF (> (ACCOUNT1 'ID) (ACCOUNT2 'ID))
      (SERIALIZED-EXCHANGE ACCOUNT2 ACCOUNT1)
      (let ((serializer1 (account1 'serializer))
    (serializer2 (account2 'serializer)))
...)))
```

This solution, by the way, assumes that only one account is *created*
at a time!  If accounts can be created in parallel, then the assigning
of account numbers must also be serialized, this way:

```
(define make-account-and-serializer
  (let ((next-id-number 1)
(ID-SERIALIZER (MAKE-SERIALIZER)))
    (lambda (balance)
      (define (withdraw amount) ...)
      (define (deposit amount) ...)
      ((ID-SERIALIZER
(LAMBDA ()
  (let ((balance-serializer (make-serializer))
(id-number next-id-number))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) withdraw)
    ((eq? m 'deposit) deposit)
    ((eq? m 'balance) balance)
    ((eq? m 'serializer) balance-serializer)
    ((eq? m 'id) id-number)
    (else (error ...))))
    (set! next-id-number (+ next-id-number 1))
    dispatch)))))))
```

There is a separate balance-serializer for each account, but
a single id-serializer for the entire account class.  In our
OOP notation we might say

```
(define-class (account balance)
  (class-vars (next-id-number 1)
      (id-serializer (make-serializer)))
  (instance-vars (balance-serializer (make-serializer))
(id-number #f))
  (initialize ((id-serializer
(lambda ()
  (set! id-number next-id-number)
  (set! next-id-number (+ next-id-number 1))))))
  (method (withdraw amount) ...)
  (method (deposit amount) ...))
```

EXTRA FOR EXPERTS
----------------

Mail program:

Umm, I haven't actually written this, but it's straightforward. You just have to put a database in the server. When a message is sent, instead of the server passing it on to the recipient's client right away, the server just adds it to the database. When a client connects, it can ask to retrieve all its messages.

One detail that's intellectually trivial but will take some effort is that a message has to be able to be more than one line of text.

3.33 averager

We want to implement c=(a+b)/2, but we don't have a division constraint; these are unnecessary since we can run the multiplication ones backwards. So we implement a+b = c*2:

```
(define (averager a b c)
  (let ((x (make-connector))
(y (make-connector)))
    (adder a b x)
    (multiplier c y x)
    (constant 2 y)
    'ok))
```

3.34 Louis's squarer

It would be nice if Louis's idea worked, and if the multiplier constraint were smart enough, maybe it could. We could add a clause to the COND in the multiplier's process-new-value procedure:

```
((and (has-value? product)
      (eq? m1 m2))
(set-value! m1 (sqrt (get-value product)) me))
```

Note that the EQ? test is *not* a test of numeric equality! On the contrary, in this situation we are assuming that m1 (and m2, since they're the same connector) has no numeric value. The EQ? test is asking if they're the same connector, not two connectors with the same value.

But our multiplier doesn't have this feature, so Louis's program won't work. More precisely, it'll work fine in the forward direction; if we give connector A a value, then Louis's program will give connector B the square of that value, because the multiplier will carry out the COND clause

```
((and (has-value? m1) (has-value? m2))
(set-value! product
    (* (get-value m1) (get-value m2))
    me))
```

since m1 and m2 are both attached to connector A. But we want our constraints to work backward, too; if we give B a value, the squarer should assign its square root to A. And that won't happen because (except in the special case of a multiplicand of zero) the multiplier constraint requires two known values in order to compute the third value.

3.35 Ben's squarer

```
(define (squarer a b)
  (define (process-new-value)
    (if (has-value? b)
(if (< (get-value b) 0)
    (error "square less than 0 -- SQUARER" (get-value b))
    (SET-VALUE! A (SQRT (GET-VALUE B)) ME))
(SET-VALUE! B (* (GET-VALUE A) (GET-VALUE A)) ME)))
  (define (process-forget-value)
    (FORGET-VALUE! A ME)
    (FORGET-VALUE! B ME))
  (define (me request)
    (COND ((EQ? REQUEST 'I-HAVE-A-VALUE)
   (PROCESS-NEW-VALUE))
  ((EQ? REQUEST 'I-LOST-MY-VALUE)
   (PROCESS-FORGET-VALUE))
  (ELSE (ERROR "UNKNOWN REQUEST -- SQUARER" REQUEST))))
  (CONNECT A ME)
  (CONNECT B ME)
  me)
```

3.36 environments

The constraint system is an OOP system, in which the classes are "connector" and the various constraint types ("adder" etc.). So we should expect to see four levels of environment: global, class, instance, and method.

But this particular program is simpler, in that there are no class variables or class methods. If, for example, the connector class had a class variable FOO, the code would be

```
(define make-connector
  (let ((foo 87))
    (lambda () ...)))
```

Instead we have the straightforward

```
(define (make-connector) ...)
```

which doesn't generate a class frame.

Make-connector and set-value! are procedures whose right bubble points to the global environment.

The first call to make-connector established a new frame:

E1: <empty>; points to global environment

The LET inside make-connector creates a procedure:

P1: params: value, informant, constraints; body (define...)...; points to E1.

and another frame:

E2: value = #F, informant = #F, constraints = (); points to E1.

The four definitions in the LET body generate

P2: params: newval, setter; body (cond...); points to E2.
P3: params: retractor; body (if...); points to E2.
P4: params: <none>; body (if...)...; points to E2.
P5: params: request; body (cond...); points to E2.

and also add four new bindings to E2:

E2: value = #F, informant = #F, constraints = (),
    set-my-value = P2, forget-my-value = P3, connect = P4, me = P5;
    points to E1.

Make-connector returns the binding for ME, namely P5, so the (define a ...) adds the binding A = P5 to the global environment.

Similarly, the (define b (make-connector)) gives another five new procedures and two new frames, which I'm going to ignore because they don't do anything interesting in the SET-VALUE! call. But we'll name them P6-P10 and E3-E4.

Now we call set-value!, which is defined in the global environment, and whose right bubble points to the global environment. So this creates

E5: connector = P5, new-value = 10, informant = user; points to global env.

The body of set-value! first calls (connector 'set-value!), i.e., it calls P5.
This creates another frame:

E6: request = set-value!; points to E2.

This call returns the value of set-my-value in E2, namely P2. We then do the outer invocation in set-value!'s body, which is in effect (P2 10 'user), so this creates

E7: newval = 10, setter = user; points to E2.

In this environment we evaluate the body of set-my-value, which is a COND expression. We start by calling has-value?, which is a global procedure, so this makes the frame

E8: connector = P5; points to global env.

This calls P5:

E9: request = has-value?, points to E2.

P5 (i.e., ME) checks the value of INFORMANT, which it finds in E2. It's false, so P5 returns #F. This is used as the argument to the primitive procedure NOT, which doesn't affect the environment, but returns #T.
Therefore we carry out the rest of the first COND clause.

* Change the binding of VALUE in E2 to 10.
* Change the binding of INFORMANT in E2 to USER.

And finally we can call for-each-except, so we're finished doing what the question asked. The actual argument values to for-each-except are USER, the inform-about-value procedure (which is defined globally), and the empty list.

3.37 expression-oriented constraint language

This is pretty straightforward, as long as you remember that C- has to use an adder constraint, not a (nonexistent) subtracter constrant, etc.

```
(define (c- x y)
  (let ((z (make-connector)))
    (adder y z x) ;; note the order!
    z))

(define (c* x y)
  (let ((z (make-connector)))
    (multiplier x y z)
    z))

(define (c/ x y)
  (let ((z (make-connector)))
    (multiplier y z x)
    z))

(define (cv num)
  (let ((z (make-connector)))
    (constant num z)
    z))
```