CHAPTER 1

Seamless. Transparent network of objects

Seamless implements Basys network to organize transparent communication between distributed objects.

SeamlessNetwork instance should be created on client and server.

```
network := SeamlessNetwork new.
```

To accept connections network should start server:

```
network startServerOn: 40422. "it is part of Basys network API"
```

Client can connect to server and retrieve remote environment:

```
remotePeer :=  network remotePeerAt: (TCPAddress ip: #[127 0 0 1]
    port: 40422).
remoteSmalltalk := remotePeer remoteEnvironment.

"or there is short version"

remoteSmalltalk := network environmentAt: (TCPAddress localAt:
    40422).
```

Returned result is proxy which delegates any received message to remote object. Remote messages are executed on server which return result back to client. Result can be returned as another proxy or as copy which contains another proxies.

In example result is reference to remote Smalltalk instance. It can access globals from remote environment:

```
remoteTranscript := remoteSmalltalk at: #Transcript.
remoteTranscript open; show: 'remote message'; cr
```

It will open transcript on server and print text on it.

Arguments of remote message are transferred to server with same logic as message result transferred to client. On server arguments can include proxies and server can send messages to them:

```
remoteTranscript print: #(1 2 3 4 5)
```

Here array will be passed to server as reference. Then on server transcript will interact with it to print it. And as result client will receive messages from server.

Seamless allows scripts evaluation on remote side of peers:

```
remotePeer evaluate: [1 + 2]. "==>3"
```

Given block is transferred to remote side and evaluated. Result is returned to client. As in other cases it could be proxy or normal object.

Proxy could load local copy by special message #asLocalObject:

```
remoteRect := remotePeer evaluate: [0@0 corner: 2@3].
    "==>aSeamlessProxy"
remoteRect asLocalObject "==> 0@0 corner: 2@3"
```

All global references of block (literals) are transferred in the way that on remote side they will be same local globals. Following example will show notification on remote image:

```
remotePeer evaluate: [Object inform: 'message from remote image'].
```

Also blocks could reference objects from local context like temps or workspace variables. They are transferred according to own strategies (explained at 1.1):

```
| result |
result := OrderedCollection new.
remotePeer evaluate: [100 to: 500 do: [:i |
      result add: i factorial ]].
```

It could lead to backward messages from server to client:

```
| rect |
rect := 0@0 corner: 2@3.
remotePeer evaluate: [rect area]. "==>6"
```

Given block is executed on server which produces message #area to client object ”rect”. Result will be first transferred to server and then from server it will be transfered back to client.

Remote blocks could perform non local return in regular Smalltalk semantics:

```
remotePeer evaluate: [1 to: 10 do: [:i | i>5 ifTrue: [^i] ] ]. "==>6"
```

Seamless also supports async evaluation of blocks without waiting result:

```
| result |
result := OrderedCollection new.
remotePeer evaluateAsync: [result add: 1000 factorial]. "it will not
    wait result"
```

1.1 Transfer strategies

By default objects transferred by reference. But some objects like numbers and strings are transferred by value. Classes can override this behaviour by implementing #seamlessDefaultTransferStrategy.

For example Object returns SeamlessTransferByReferenceStrategy. But numbers and strings return SeamlessTransferByValueStrategy.

```
Object>>seamlessDefaultTransferStrategy
  ^SeamlessTransferStrategy defaultByReference

Number>>seamlessDefaultTransferStrategy
  ^SeamlessTransferStrategy defaultByValue

MessageSend>>seamlessDefaultTransferStrategy
  ^SeamlessTransferStrategy defaultForMessageSend
```

Default strategies are defined as singletons to reduce garbage during transfer.

Default strategies can be overridden on network level. Strategy can be explicitly added to network for specific set of objects.

```
network addTransferStrategy: (SeamlessTransferByReferenceStrategy
    for: (Instance of: Point).
"or short versions:"
network transferByReference: (Kind of: Class).
network transferByValue: (Kind of: Point).
```

Set of objects are defined by criteria which understands #matches: message. In this example StateSpecs is used.

Transfer by reference

For objects which are transferred by reference network creates SeamlessObjectReference and transmits it over network instead of real object. On server side network creates proxies for received references.

By default proxy is SeamlessProxy which delegates all messages to remote side. But it is possible to use specific proxy representation for distributed objects.

New kind of reference should be created by subclassing SeamlessObjectReference. It should override #createProxy method to return required proxy representation.

Then required classes can return new kind of reference from method #createSeamlessReference.

For example look at existing implementations:

```
Object>>createSeamlessReference
  ^SeamlessObjectReference new

SeamlessObjectReference>>createProxy
  ^SeamlessProxy for: self

SeamessSyncRequestContext>>createSeamlessReference
    ^SeamlessRequestContextReference new

SeamlessRequestContextReference>> createProxy
    ^SeamlessRemoteContext new
```

SeamlessObjectReference supports caching. SeamlessTransferByReferenceStrategy could be created with set of messages which result should be transferred together with given object.

When SeamlessProxy receives message it first check cache for it. And if message is cached then existing result will be returned. No remote communication will be performed.

Such strategy could be added to network by:

```
network transferByReference: (Kind of: Class) withCacheFor: #(name
    allInstVarNames).
```

Transfer by value

This strategy transfers object as copy. It is not specified what should happens with it instance variables.

For example Rectangle can be transferred by value with two different ways:

```
originReference corner: cornerReference
```

In this case both corners was transferred by reference.

But it can be:

```
10@20 corner: cornerReference
```

Which means that for origin point Seamless chooses transfer by value strategy but for corner it chooses transfer by reference.

Objects could specify concrete meaning of what is value. For example OrderedCollection will also transfer by value internal array. To specify it method #prepareValueTransferBy: should be implemented:

```
OrderedCollection>>prepareValueTransferBy: aSeamlessObjectTransporter
  aSeamlessObjectTransporter transferAsValue: array
```

Transfer by deep copy

This strategy transfers object as deep copy. Any defined strategies for internal state of object are ignored.

Logic for object transfer as whole is delegated to concrete serialization protocol. In case of Fuel it will just write given object into stream without any Seamless analysis.

Transfer by referenced copy

This strategy will transfer object by reference but on remote side copy of object will be used as representation.

When copy will return back to client original object will be used.

SeamlessObjectCopyReference is used here as special reference to be sent between peers.

1.2 SeamlessObjectTransporter

Seamless delegates object sending and receiving to SeamlessObjectTransporter. It is abstract class which implements general behaviour to prepare sent and received objects. Subclasses use prepared information to correctly implement concrete transfer protocol:

```
transporter transferObject: anObject by: aSocket
transporter receiveObjectBy: aSocket
```

Currently Fuel protocol is implemented by SeamlessFuelObjectTransporter.

Transfer objects

Before sending object transporter scans full object graph and collects nodes which should be transferred by reference and by value

```
SeamlessObjectTransporter>>prepareObjectForTransfer
  | strategy |
  objectsByReference := IdentityDictionary new.
  objectsByValue := IdentitySet new.
  traveler := ObjectTraveler on: object.
  traveler referencesDo: [:eachNode |
        (objectsByValue includes: eachNode) ifFalse: [
          strategy := network transferStrategyFor: eachNode.
          strategy prepareTransferOf: eachNode by: self]]
```

It asks transfer strategy for each node. And strategy commands what to do with it:

```
TransferByReferenceStrategy>>prepareTransferOf: anObject by:
    aSeamlessObjectTrasporter
  | reference |
  reference := aSeamlessObjectTrasporter transferAsReference:
    anObject.
  cachedMessages do: [ :each |
    reference cacheMessage: each with: (anObject perform: each)]

TransferByValueStrategy>>prepareTransferOf: anObject by:
    aSeamlessObjectTrasporter
  "By default transporter transfer given object by value. So we
    don't need to do anything here.
  But we allow objects to specify default meaning of value transfer.
  For example Object do nothing.
  But OrderedCollection marks internal array to be transfered as
    value too"
  anObject prepareValueTransferBy: aSeamlessObjectTrasporter

TransferByReferencedCopyStrategy>>prepareTransferOf: anObject by:
    aSeamlessObjectTrasporter
  aSeamlessObjectTrasporter transfer: anObject asReference:
    [SeamlessObjectCopyReference to: anObject]
```

At the end concrete transporter uses collected information to substitute required objects by references.

```
SeamlessFuelObjectTransporter>>transferObjectBy: aSocket
  | stream serializer reference |
  stream := network transferWriteStreamOn: aSocket.

  serializer := FLSerializer newDefault.
  serializer analyzer
        when: [ :o | (reference := objectsByReference at: o
    ifAbsent: [ nil ]) notNil ]
        substituteBy: [ :o | reference ].
   serializer serialize: object on: stream.
  stream flush
```

Receiving objects

Concrete transporters implement reading objects from network.

```
FuelObjectTransporter>>receiveObjectBy: aSocket
  | stream |
  stream := network transferReadStreamOn: aSocket.
  [
    object := (FLMaterializer newDefault materializeFrom: stream)
    root.
  ] on: FLError do: [ :err | BasysWrongProtocolError signalAs: err ]
```

Transporter can detect wrong data format and signal failure. Basys will catch it and close "alien" connection.

When object is received transporter prepare it for external usage. It scans full object graph to replace all seamless references by local objects or proxies

```
SeamlessObjectTransporter>>prepareReceivedObjectWhichWasFrom:
    aRemotePeer
  traveler := ObjectTraveler on: object.
  traveler skip: aRemotePeer.
  traveler referencesDo: [:each |
    each representYourselfIn: self forTransferFrom: aRemotePeer.
  ].
  ^object

Object>>representYourselfIn: anObjectTransporter forTransferFrom:
    aRemotePeer
  "some objects could have specific remote representation"

SeamlessObjectReference>>representYourselfIn: anObjectTransporter
    forTransferFrom: aRemotePeer
  senderPeer := aRemotePeer.
  anObjectTransporter materializeReceivedReference: self

SeamlessObjectTransporter>>materializeReceivedReference:
    aSeamlessObjectReference
  | existingObject |
  existingObject := network objectFor: aSeamlessObjectReference
    ifNotNew: [:o | traveler skip: o].
  self replaceReceivedObjectBy: existingObject.
  traveler alsoLookAt: aSeamlessObjectReference
```

Method #representYourselfIn:forTransferFrom: provides extra hooks for transferring objects.

For example there is SeamlessObjectContainer. It is always transferred by value together with content. On remote side content will substitute container:

```
SeamlessObjectContainer>>representYourselfIn: anObjectTransporter
    forTransferFrom: aRemotePeer
  anObjectTransporter replaceReceivedObjectBy: content
```

It allows send message arguments with forced value strategy:

```
remotePoint distanceTo: (SeamlessObjectContainer with: 2@3)
"or short version"
remotePoint distanceTo: (2@3) asTransferredByValue
```

Independently of transfer strategy defined for given point it will be transferred by value as argument of #distanceTo: message.

1.3 Managing distributed objects

SeamlessDistributedObjects is implemented to organize thread safe access for distributed objects.

It implements mapping between objects and their references.

Reference is represented by SeamlessObjectReference subclasses. It contain id, ownerPeerId and senderPeer instance variables.

Pair "id and ownerPeerId" is globally unique and used as identity property.

senderPeer instance variable contains sender of reference.

SeamlessDistributedObjects provides access to object by reference and it can find reference for object. SeamlessNetwork implements convenient methods for this:

```
SeamlessNetwork>>objectFor: aSeamlessObjectReference
  ^distributedObjects
    at: aSeamlessObjectReference
    ifAbsentUseProxy: [ aSeamlessObjectReference createProxy]
```

It creates proxy if no local object exists for given reference.

```
SeamlessNetwork>>referenceFor: anObject
  ^self referenceFor: anObject ifNewUse: [ anObject
     createSeamlessReference]
```

It creates reference for new distributed object. Created reference will contain local peer as sender-peer.

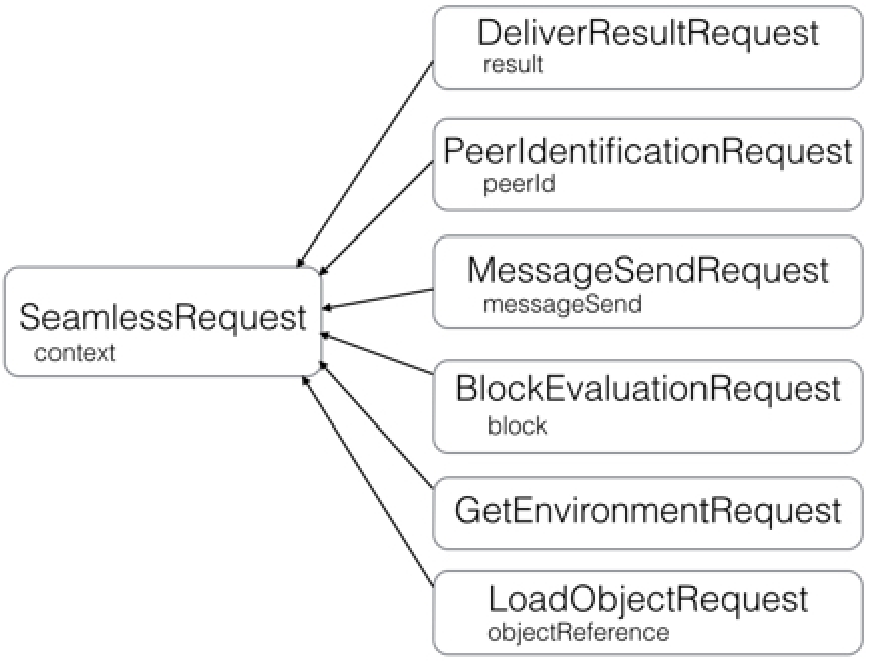
Now no garbage collection implemented for distributed objects. They could be cleaned manualy by ”network destroy” command.

1.4 Seamless requests

Seamless peers communicate with each others by sending request objects.

```
peer sendDataPacket: (SeamlessMessageSendRequest with: aMessageSend)
```

It is subclasses of SeamlessRequest. Remote message send is just one type of it:



Sending request to peer not waits any result. It just sends serialized data by free connection.

To perform synchronous request it should be sent by special context object:

```
context := peer createSyncRequestContext.
returnedValue := context sendRequest: (SeamlessMessageSendRequest
    with: aMessageSend)
```

Context represents logical request sender. Request holds it in context variable and together they are transferred over network.

Here is SeamlessSyncRequestContext. It forks request sending and waits result.

Requests should implement method #executeFor: which are called on receiver-peer as part of network data processing.

During execution requests can return result to sender.

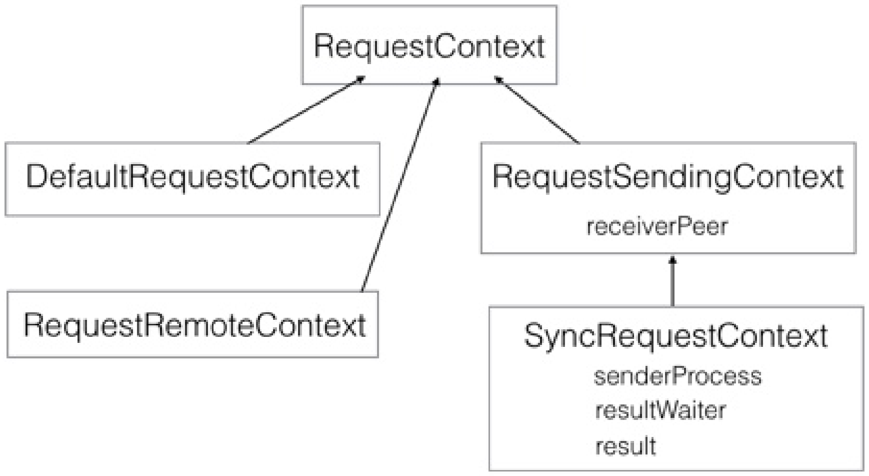
Request result is represented by SeamlessRequestResult subclases:

- SeamlessReturnValueResult
- SeamlessThrowExceptionResult

Requests should return result by context:

```
SeamlessMessageSendRequest>>executeFor: senderPeer
  | messageResult requestResult |
  [
    messageResult := messageSend value.
    requestResult := messageResult asSeamlessRequestResult.
  ] on: Exception - Halt - Notification do: [ :err | requestResult
    := err asSeamlessRequestResult ].

  context return: requestResult to: senderPeer

SeamlessPeerIdentificationRequest>>executeFor: senderPeer
  | identifiedPeer |
  identifiedPeer := senderPeer beIdentifiedAs: peerId.

  context returnValue: senderPeer localPeerId to: identifiedPeer

SeamlessGetEnvironmentRequest>>executeFor: senderPeer
  context returnValue: Smalltalk to: senderPeer
```

There are few types of contexts. They all implement method #return:to: to send result to sender.



Request sending contexts are used to send requests to receiver-peer and somehow return result from it. They are subclasses of SeamlessRequestSendingContext.

They should implement #sendRequest: method and simple version of #return: without senderPeer argument.

For example look at sync context:

```
SeamlessSyncRequestContext>>sendRequest: aSeamlessRequest
  aSeamlessRequest context: self.
  self forkProcessingOf: aSeamlessRequest.
  resultWaiter wait.
  ^result returnValue

SeamlessSyncRequestContext>>return: aRequestResult
    result := aRequestResult
    resultWaiter signal
```

When request is executed on receiver-peer, the sender-peer should receive result and somehow call #return: method of waiting sync context.

To implement it special trick is used.

Sync context is always transferred by reference and on receiver-peer it is represented by special proxy SeamlessRemoteContext. As proxy it can delegate messages to real remote object which is sync context in that case. So to implement return SeamlessRemoteContext just sends remote message #return: to itself. It is performed by special SeamlessDeliverResultRequest:

```
SeamlessRemoteContext>>return: result to: senderPeer
  senderPeer sendDataPacket: (SeamlessDeliverResultRequest result:
    result to: self)

SeamlessDeliverResultRequest>>executeFor: senderPeer
  context return: result
```

SeamlessDeliverResultRequest is executed on sender side where context is sync context which is waiting on semaphore for result.

1.5 SeamlessLogger

SeamlessLogger is special tool to analyse network communication. It needs to be started:

```
SeamlessLogger start
```

Logger will keep log of all transferred requests over network. Any incoming or outgoing requests will be immedietely printed to Transcript. To pause logger it should be stopped:

```
SeamlessLogger stop
```

It not cleans requests history. It just disables requests interception.

Logger can be stopped and cleaned by:

```
SeamlessLogger stopAndClean
```

Or it could be restarted with fresh history:

```
SeamlessLogger startAfresh
```

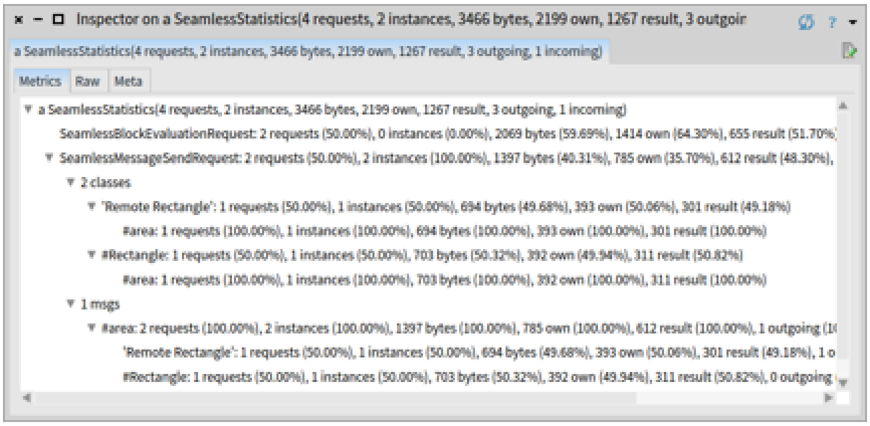
So after starting logger transcript will show requests. But most interesting part of logging system is analysis of full communication statistics:

```
stat := SeamlessLogger collectStatistics
```

Statistics could be inspected with nice GT extensions. For example statistics on following code:

```
remoteRect := remotePeer evaluate: [0@0 corner: 2@3].
remoteRect area. "==>6".
localRect := 0@0 corner: 5@2.
remoteRect evaluate: [remoteRect area + localRect area] "==>16"
```

will show number of messages, receivers and bytes which was transfered over network in dimension of receiver class or message selector:


Inspector on a SeamlessStatistics(4 requests, 2 instances, 3466 bytes, 2199 own, 1267 result, 3 outgoir
a SeamlessStatistics(4 requests, 2 instances, 3466 bytes, 2199 own, 1267 result, 3 outgoing, 1 incoming)
Metrics
Raw
Meta

```
▼ a SeamlessStatistics(4 requests, 2 instances, 3466 bytes, 2199 own, 1267 result, 3 outgoing, 1 incoming)
    SeamlessBlockEvaluationRequest: 2 requests (50.00%), 0 instances (0.00%), 2069 bytes (59.69%), 1414 own (64.30%), 655 result (51.70%)
  ▼ SeamlessMessageSendRequest: 2 requests (50.00%), 2 instances (100.00%), 1397 bytes (40.31%), 785 own (35.70%), 612 result (48.30%),
    ▼ 2 classes
      ▼ 'Remote Rectangle': 1 requests (50.00%), 1 instances (50.00%), 694 bytes (49.68%), 393 own (50.06%), 301 result (49.18%)
          #area: 1 requests (100.00%), 1 instances (100.00%), 694 bytes (100.00%), 393 own (100.00%), 301 result (100.00%)
      ▼ #Rectangle: 1 requests (50.00%), 1 instances (50.00%), 703 bytes (50.32%), 392 own (49.94%), 311 result (50.82%)
          #area: 1 requests (100.00%), 1 instances (100.00%), 703 bytes (100.00%), 392 own (100.00%), 311 result (100.00%)
    ▼ 1 msgs
      ▼ #area: 2 requests (100.00%), 2 instances (100.00%), 1397 bytes (100.00%), 785 own (100.00%), 612 result (100.00%), 1 outgoing (1
          'Remote Rectangle': 1 requests (50.00%), 1 instances (50.00%), 694 bytes (49.68%), 393 own (50.06%), 301 result (49.18%), 1 o
          #Rectangle: 1 requests (50.00%), 1 instances (50.00%), 703 bytes (50.32%), 392 own (49.94%), 311 result (50.82%), 0 outgoing
```

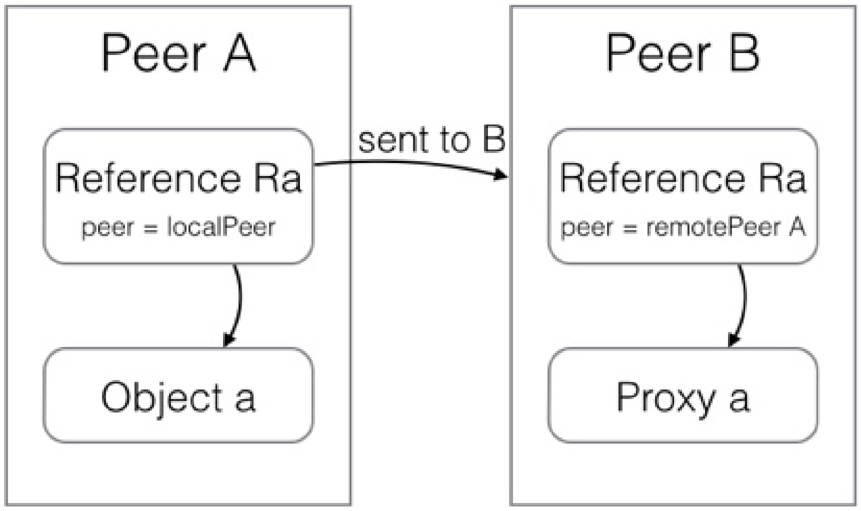
1.6 Future work

Garbage collection

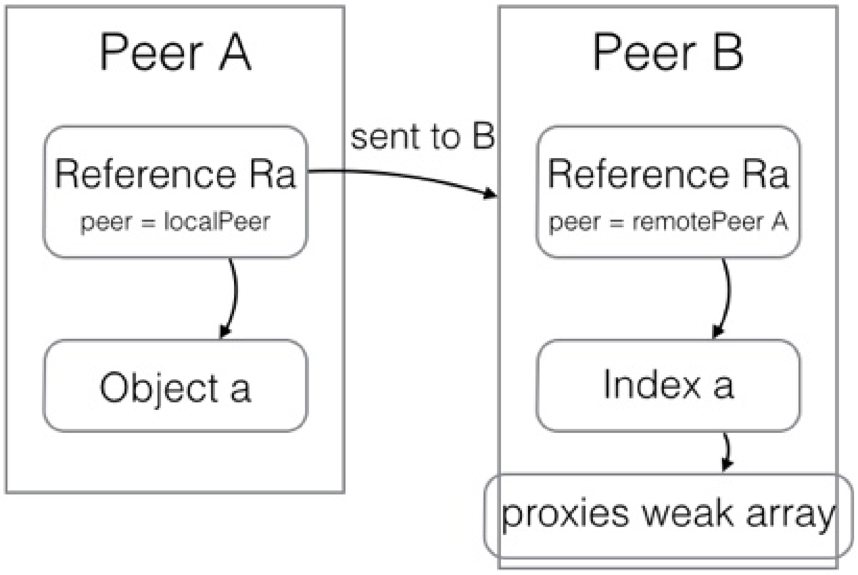
There is no garbage collection of distributed objects yet. There are three cases:

- cleaning unused proxies
- cleaning unused real objects (distributed objects)
- cleaning unused peers

Cleaning unused proxies

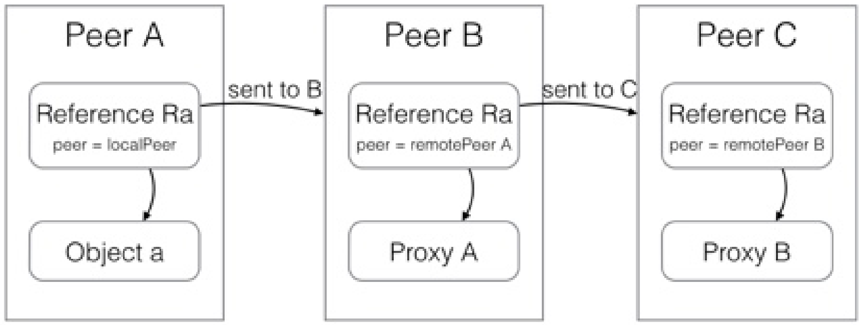


First case can be implemented by extracting all proxies to separate weak collection inside SeamlessDistributedObjects structure. Currently this container use two dictionaries: referencesToObject and objectsToReferences, which used to manage proxies and real objects equivalently.



Container can hold integers for proxies which will point to index inside separated weak array. This will allow garbage collector remove unused proxies. And by this Seamless can cleanup unused references.

It works fine for two peers. But when reference is shared between multiple peers such approach has problems

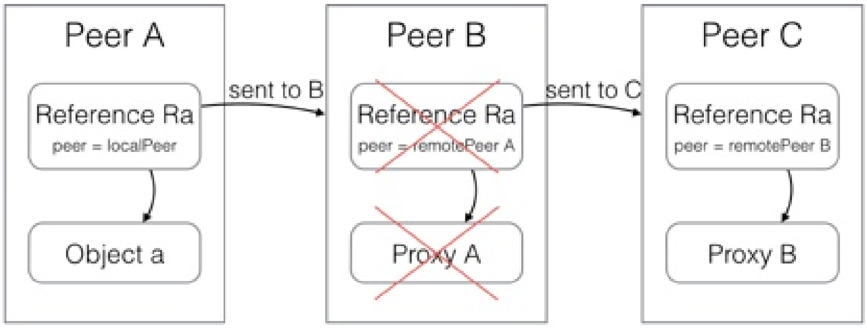


Here example of network with three peers A, B, C.

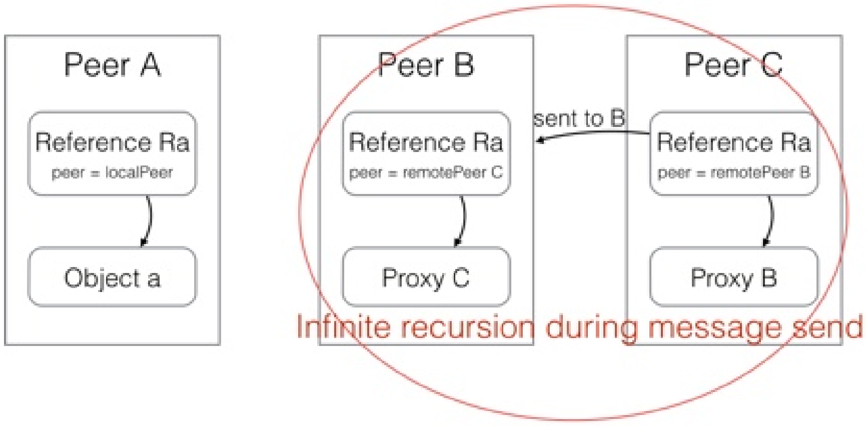
Peer A transfer reference Ra to peer B. On peer B reference will be presented by proxy A which will delegates all messages to peer A.

Then peer B transfers this reference to peer C. On peer C reference will be presented by proxy B which will delegates all messages to peer B which will resent it to peer A.

Now imaging that reference Ra not used anymore on peer B. And peer B removed it from own distributed objects.



What will happen if peer C will send messages to proxy B?



Proxy B will resend message to peer B. But peer B not anymore has reference Ra. So peer B will create proxy C for it which will point to peer C. And proxy C will resent message back to peer C. Which will lead to infinite recursion.

Possible solution to this problem is to forbid remote indirect messages. Seamless can always try to find owner peer of reference on current network. And if it not found error can be signalled. Otherwise Seamless can send message directly to owner peer instead of indirect transfer through sender-peer.

Such restriction can be more stronger. Seamless can forbid transfer of references o peers which not know owner peers. In that case sending objects which just contains references can be failed. No matter if this references will never receive messages on this peers.

With that approach concrete application will be responsible to ensure that peers know each others.

In the case of two peers network this restrictions make no sense. And also classic client server application should not suffer by this. Usually client data not becomes shared between other clients.

By the way guarantee of direct messages will provide better performance for distributed applications.

Cleaning unused distributed objects

When object are going to be transfered over network it become distributed. Special reference object created for him to represent it inside network. Seamless can't just remove distributed object when it is not anymore used on owner peer. Other peers can have reference to it which allow them to send messages to it.

So secure way to clean unused distributed objects is to ask all known peers about it:

- all local references can be send to knowk peers to detect what is not used
- all known peers can be asked for used references to detect local references which is really not used in network.

But known peers can be unavailable at garbage collection time. What to do in that case?

Different application can put differen requirements for such cases. Some of them can just say: remove unavailable peers and continue garbage collection. Others can say: try again after hour and then remove it.

So it is responsibility of application to put specific strategy on garbage collection of distributed object.

Seamless should implement general requests for garbage collection with hooks which will provide required flexibility.

Cleaning unused peers

BasysNetwork contains remote peers collection. Now nobody clean it automatically. So any connected client keep in network forever.

Solution is using weak collection for remote peers. It is not problem if server will remove particular remote peer but client will use it in future. Client always can establish new connection and server will recreate peer for it.

Special disconnecting request can be implemented to explicitly remove peers and release all it resources.

Optimizing fuel protocol

Analysis of transferred object graph can be merged with fuel analyses to remove duplicated object traversal

Distributed exceptions

It should be possible to implement normal Smalltalk semantics for remote exception handling.

Now in case of exception remote message send or evaluated block will terminate remote process and return special SeamlessRemoteException with string information about original error.

On sender side it will open debugger with this general exception on local process (which sent request). No remote stack will be shown.

Advanced system should return signalled error to sender as normal exception with references to remote signaler context (and process). Then exception could be handled with normal #on:do: message.

And in case of unhandled error debugger will show local and remote stack together which will lead to regular IDE experience.

Also it will allow resumable exceptions.

Better integration with debugger

It could be possible to extend debugger such way that step into remote message on sender side will show remote context from receiver and allow debug it deeply.

And vice versa debugging process of incoming remote message on receiver could show remote sender stack together with local.