THE POWER OF FAR MONITORS

Christian Maurer

Institut für Informatik
Fachbereich Mathematik und Informatik
Freie Universität Berlin

10.5.2017

Abstract. As a continuation of the technical report "Universal synchronization objects" we present a variant of its "monitor"-concept introducing monitor functionality over the net. Among other applications (as e.g. the left right problem) it allows elegant implementations of several classical distributed algorithms as e.g.
– network traversal by depth-first- and breadth-first-search,
– computation of the topology of a local network and
– election of a leader in a local network.

Introduction

Let us quote the last sentence in the abstract of the old report[1]:

> "There is strong evidence that central aspects of other synchronization mechanisms such as message passing could be condensed in similar objects; in case this conjecture turns out to be true, the client server paradigm will be examined in a subsequent paper."

In this paper we show, that it is indeed possible to construct a generalization of the universal monitor concept to provide the handling of monitor calls over network channels, making things much easier than constructing remote procedure calls.

It should be noted, that meanwhile all examples of the quoted report are ported from Modula-2 via Java to Go[2]. This language is indeed very useful, because its sophisticated design—combining static type checking with dynamic typing at run time—is tremendously helpful particularly from the point of view of software engineering. Among other advantages it allows a strict separation between specifications and implementations of abstract data types and—by means of the interface concept—the use of a sort of inheritance at the level of specifications, which IMHO is more important than at the level of implementations.

First of all some type definitions have to be introduced, contained in the package `obj` of murus[3]:

```
package obj

type (
  Any interface{}
  Op func (Any)
  FuncSpectrum func (Any, uint) Any
  PredSpectrum func (Any, uint) bool
)
```

With these definitions we give the specification of our data type "far monitor":

Key words and phrases. breadth first, channel, concurrency, depth first, distributed, graph, leader election, monitor, network, neighbour, process, ring, search, TCP/IP, topology, traversal, tree.

[1] ftp://ftp.inf.fu-berlin.de/pub/reports/tr-b-00-09.ps.gz
[2] http://golang.org
[3] https://murus.org

Typeset by $\mathcal{AMS}$-TEX

```
package fmon

import (. "murus/obj"; "murus/host")

type
  FarMonitor interface { // x always denotes the calling object.

// Pre: i < number of monitor functions of x.
// The calling process was delayed, if the predicate of x
// for the i-th monitor function is not true.
// The monitor function fs(_, i) is executed on the server;
// if necessary, the calling process was delayed until ps(a, i) == true.
// The value of a is sent to the server with this call.
// It returns the value (of the type of x), that the server returns.
  F (a Any, i uint) Any

// All net channels used by x are closed.
  Fin()
}

// Pre: fs and ps are defined in their second argument for all i < n.
//      h is by an entry in /etc/hosts or DNS-lookup reachable,
//      p is not used by any network service.
//      If s == true, New is called in a process on the host h.
// Returns a new far monitor with n monitor operations.
// Its type is the type of a, i.e. objects of this type are passed
// between server and clients.
// For a == nil the server returns upon a monitor call a byte stream
// (object of type []byte); in this case the caller is responsible
// for decoding that stream into an object of the type of the object,
// that he sent as first argument in his monitor function call F.
// fs(_,i) for i < n is the i-th monitor function and ps(_, i) is the
// corresponding predicate determining whether the monitor functions
// can be executed.
// h is the server executing the monitor calls and p is the port
// used by the TCP-IP connection between the server and the clients;
// the needed net channels are opened.
// The far monitor runs as server, iff s == true; otherwise as client.
func New (a Any, n uint, fs FuncSpectrum, ps PredSpectrum,
          h host.Host, p uint16, s bool) FarMonitor {
  return new_(a, n, fs, ps, h, p, s)
}

// See above. Additionally, st is executed by the server before it starts serving.
func NewS (a Any, n uint, fs FuncSpectrum, ps PredSpectrum,
           h host.Host, p uint16, s bool, stmt Stmt) FarMonitor {
  return newS(a, n, fs, ps, h, p, s, stmt)
}
```

The implementation makes use of message passing by using channels—an essential feature of the Go language, adopted from HOARE's CSP (see [H1]). The reason for this is what we consider a big advantage for our concept: the possibility of using guarded selective waiting—a technique adopted from Pascal-FC by BURNS/DAVIES ([BD]) and from ADA.
It is implemented in Go by our `When`-construct (to be found also in the package `murus/obj`). The idea is—by specification of the language—that a `select` on a `nil`-channel blocks:

```
func When (b bool, c chan Any) chan Any {
  if b {
    return c
  }
  return nil
}
```

Here is the implementation of our abstract data type `FarMonitor`:

```
package fmon
import (. "murus/ker"; . "murus/obj"; "murus/host"; "murus/nchan")
type farMonitor struct {
                       Any "type of objects the monitor functions operate on"
                       uint "number of monitor functions"
                     ch []nchan.NetChannel
                       FuncSpectrum; PredSpectrum
                       bool "true iff the monitor is a server"
                       }

func new_(a Any, n uint, fs FuncSpectrum, ps PredSpectrum,
          h host.Host, p uint16, s bool) FarMonitor {
  return newS (a, n, fs, ps, h, p, s)
}

func newS (a Any, n uint, fs FuncSpectrum, ps PredSpectrum,
           h host.Host, p uint16, s bool) FarMonitor {
  if n == 0 { Panic ("fmon.New must be called with 2nd arg > 0") }
  x := new(farMonitor)
  x.Any, x.uint, x.bool = Clone(a), n, s
  x.ch = make([]nchan.NetChannel, x.uint)
  in, out := make([]chan Any, x.uint), make([]chan Any, x.uint)
  any, ok := make([]Any, x.uint), make([]bool, x.uint)
  for i := uint(0); i < x.uint; i++ {
    x.ch[i] = nchan.NewCS (x.Any, h, p + uint16(i), s)
    in[i], out[i] = x.ch[i].Chan()
  }
  if ! x.bool { return x } // x is a client
  x.FuncSpectrum, x.PredSpectrum = fs, ps
  for i := uint(0); i < x.uint; i++ {
    go func (j uint) {
      for {
        select {
        case any[j], ok[j] = <-When (x.PredSpectrum (x.Any, j), in[j]):
          if ok[j] {
            if x.PredSpectrum (any[j], j) {
              any[j] = x.FuncSpectrum (any[j], j); out[j] <- any[j]
            } else {
              out[j] <- x.Any
            }
          }
        default:
        }
        Msleep(300)
      }
    }(i)
  }
  return nil
}

func (x *farMonitor) F (a Any, i uint) Any {
  if x.ch[i] == nil { Panic("fmon.F: x.ch == nil") }
  x.ch[i].Send (a)
  return x.ch[i].Recv()
}

func (x *farMonitor) Fin() {
  for i := uint(0); i < x.uint; i++ { x.ch[i].Fin() }
}
```

The underlying package `nchan` provides the traffic over the net via TCP/IP. Its specification is:

```
package nchan

import (. "murus/obj"; "murus/host")

const Port0 = uint16(1<<16 - 1<<14) // first private port (= 49152)

type NetChannel interface { // Channels for passing objects over the net

// Pre: a is of the type of x.
// a is sent on x (resp. if x is a 1:n channel and
// the calling process is a server, on the actual subchannel of x)
// to the communication partner of the calling process.
  Send (a Any)

// Returns a slice of bytes, if x was created by New with nil as first
// argument. In this case, the client is responsible for decoding that
// slice, according to the type of what was sent.
// Otherwise, i.e. if x is bound to a type:
// Returns the object of the type of x, that was received on x (resp. if
// x is a 1:n channel and the calling process is a server, on the actual
// subchannel of x) from the communication partner, if that was received;
// returns an empty object otherwise.
// The calling process was blocked, until an object was received.
  Recv() Any

// The port used by x is not used by a network service on the calling host.
  Fin()
}

// h0 denotes the calling host (running the calling process).
// Pre: h is in /etc/hosts or resolvable per DNS (! h.Empty()).
//      me != i; me is the identity of h0 and i is the identity of h
//      (me, i < number of hosts involved).
//      p > 0; p is not used on h0 or h by a network service.
//      The communication partner calls New with
//      - an object of the same type as the type of a as 1st argument,
//      - with the host of the calling process as 4th argument.
//      - with the values of me and i reversed,
//        i.e. me/i as 3rd/2nd argument and
//      - an identical value of the 5th argument.
// Returns a asynchronous 1:1 channel for messages of the type of a
// between h0 and h over port p.
// p is now used on h0 and h by a network service.
// In case of consecutive calls of New you have to keep      //
// the correct pairing in both programs to avoid deadlocks!  //
func New (a Any, me, i uint, h host.Host, p uint16) NetChannel {
  return new_(a,me,i,h,p)
}

// See above function. To be called in the constructor of a far monitor.
// h is the server; s == true, if the calling process is the serving monitor.
// i and o are the in- and outgoing channels for the internal communication
// of the calling far monitor.
func NewCS (a Any, h host.Host, p uint16, s bool, i, o chan Any) NetChannel {
  return newcs(a,h,p,s,i,o)
}

// Pre: i, j < n, i != j.
// Returns consecutive port numbers Port0 .. Port0 + n*(n-1)/2-1.
func Port (n, i, j uint) uint16 { return port(n,i,j) }
```

The asymmetry of the used TCP/IP-operations in the package `murus/nchan` is not visible outside, but

completely hidden in that package. In case all hosts envolved are different, it is easy to decide, which of the two ends of a net channel has to play the role of the server: Simply that one with the smaller IP (by numerical comparison). Otherwise, the problem is solved by using the identities of the processes involved (which are numbered consecutively, starting with `0`).
For its implementation and for the used package `host` we refer to the corresponding packages of murus.

The left right problem

We restrict the resumption of the examples in the old report to the "left right problem". It is defined by the invariant `nL == 0 || nR == 0`, where `nL`/`nR` denote the number of active lefties/righties.
Here is the specification in Go (without those constructors, that do not matter in the current context):

```
package lr
import "murus/host"

type LeftRight interface { // protocols for the left right problem
  LeftIn ()
  LeftOut ()
  RightIn ()
  RightOut ()
}

func NewFarMonitor (h host.Host, p uint16, s bool) LeftRight { return newFMon (h, p, s) }
```

The implementation assures the invariant:

```
type farMonitor struct {
                nL, nR uint
                       fmon.FarMonitor
                       }
func newFMon (h host.Host, p uint16, s bool) LeftRight {
  x := new(farMonitor)
  ps := func (a Any, i uint) bool {
          switch i {
          case leftIn:
            return x.nR == 0
          case rightIn:
            return x.nL == 0
          }
          return true
        }
  fs := func (a Any, i uint) Any {
          switch i {
          case leftIn:
            x.nL++
          case rightIn:
            x.nR++
          case leftOut:
            x.nL--
          case rightOut:
            x.nR--
          }
          return true
        }
  x.FarMonitor = fmon.New (true, nFuncs, fs, ps, h, p, s)
  return x
}

func (x *farMonitor) LeftIn() { x.F(true, leftIn) }
func (x *farMonitor) LeftOut() { x.F(true, leftOut) }
func (x *farMonitor) RightIn() { x.F(true, rightIn) }
func (x *farMonitor) RightOut() { x.F(true, rightOut) }
```

The package is used as follows:

The program has to be started on a server with a call of the constructor `New` that is given the following parameters:

(1) the server as first,
(2) a free port as second and
(3) the value `true` as last parameter.

On a server these values have to be the same except the last one—it must have the value `false`.
Lefties enclose their critical sections than by the "protocol brackets"

```
lr.LeftIn(); ... /* critical section */ ; lr.LeftOut()
```

where `lr` is the value of the constructor (righties do that analogously).
The "bounded" left right problem can also be managed by far monitors with only a tiny bit more effort. The constructor transports the maximal numbers `mL`/`mR` of lefties/righties within the same turn:

```
func NewFarMonitorBounded (l, r uint, h host.Host, p uint16, s bool) LeftRight {
  return newFMonB(l,r,h,p,s)
}
```

and the implementation checks that:

```
type farMonitorBounded struct {
                        nL, nR, // number of active lefties/righties
                        tL, tR uint // number of lefties/righties within one turn
                               fmon.FarMonitor
                               }

func newFMonB (mL, mR uint, h host.Host, p uint16, s bool) LeftRight {
  x := new(farMonitorBounded)
  ps := func (a Any, i uint) bool {
          switch i {
          case leftIn:
            return x.nR == 0 && x.tL < mL
          case rightIn:
            return x.nL == 0 && x.tR < mR
          }
          return true
        }
  fs := func (a Any, i uint) Any {
          switch i {
          case leftIn:
            x.nL++; x.tL++; x.tR = 0
          case rightIn:
            x.nR++; x.tR++; x.tL = 0
          case leftOut:
            x.nL--
          case rightOut:
            x.nR--
          }
          return true
        }
  x.FarMonitor = fmon.New (true, nFuncs, fs, ps, h, p, s)
  return x
}
```

Readers and Writers

You can simply use the bounded left right problem with "`R`"/"`W`" instead "`L`"/"`R`" for readers and writers (alliteration ...) resp. and `mW == 1` as maximal number of active writers.
Apart from that it should be quite clear by now, that all the things about readers and writers in the old report are built analogously to the left right problem.
So we do not give any details here but simply refer to the package `murus/rw`.

Distributed graphs

Every communication problem in a network can be considered as a problem in a graph: The vertices are the processes involved and the edges are the communication channels between the hosts on which the processes are called.
So it should be quite clear, that—assuming the existence of an abstract data type "graph"—the concept of a "distributed graph" makes use of that.

```
package dgra

import ("murus/gra"; "murus/host")

type TopAlg byte; const ( // computation of the net topology
  ... = TopAlg(iota)
  FmMatrix; FmGraph
)
type ElectAlg byte; const ( // election of a leader in a ring
  ... = ElectAlg(iota)
  ...; FmDFSE
)
type TravAlg byte; const ( // traversal of the net
  ... = TravAlg(iota)
  FmDFSA; FmBFS; FmDFSRing
)
type DistributedGraph interface {

  gra.Graph

// Pre: hs must have been globally set to avoid conflicts.
// The hs are the hosts of x.
  SetHosts (hs []host.Host)

// r is the root of x.
  SetRoot (r uint)

  ... several more methods
}

// Returns a new distributed Graph with underlying Graph g.
func New (g gra.Graph) DistributedGraph { return new_(g) }
```

The types in the top of the specification show that we address three classes of problems:

(1) the computation of the topology of a network,
(2) the election of a leader in a network and
(3) the traversal of a network (with some function to operate on the vertices).

The starting "`Fm`" in the names denotes implementations of algorithms based on far monitors (there are some others in the package not mentioned here).
The processes are uniquely identified by natural numbers $0, \ldots, n-1$ (n = number of processes). If they run on different hosts, of course it has to be made sure that these hosts correspond to their identities in the call of `SetHosts`.
The graph of a process participating in a distributed algorithm is at the beginning the "star" in the global graph, consisting of its vertex and the edges to (or from) its neighbours (but of course not any edge between its neighbours). So the process has no global knowledge whatsoever of the whole graph—neither the number of processes in it nor any details of processes or hosts "behind" its star.
This has to be respected by an implementation: The global graph has to be constructed it each process has to be assigned the host it runs on; but in the call of the constructor `New` only the star of the calling process and in the call of `SetHosts` only its neighbours are used as arguments (the `murus/gra`-package provides the method `Star` for that purpose which allows the use of identical source code for all processes.)
The call of `SetRoot` determines the process that acts as "environment".
Here is the construction of the internal type representation:

```
import (
  . "murus/obj"; "murus/host"; "murus/nchan"; "murus/fmon"
  "murus/vtx"; "murus/edg"; "murus/adj"; "murus/gra"
)
type distributedGraph struct {
                            gra.Graph // neighbourhood of the current vertex
                                      // must not be changed by any application
                  tmpGraph, // a clone of gra.Graph; these three graphs are only
                tree, ring gra.Graph // temporary graphs for the algorithms to work with
                           bool "Graph.Directed"
                             n, // number of neighbours of the current vertex
                            me, // identity of the current process
                          root uint
                 actVertex, // the vertex in the Graph representing the actual process
                 tmpVertex vtx.Vertex
                   tmpEdge edg.Edge
                   actHost host.Host // localhost
                        nb []vtx.Vertex // the neighbour vertices
                      host []host.Host // the neighbour hosts
                        nr []uint // and their identities
                        ch []nchan.NetChannel // the channels to the neighbours
                   visited []bool
                       mon []fmon.FarMonitor
                      monM []fmon.FarMonitorM
                    parent uint
                     child []bool
               time, time1 uint
                           port,
          srvMport, clMport []uint16
                            top adj.AdjacencyMatrix
                           rank uint // unfortunately a dirty trick must be used
  diameter, distance, leader uint
                           TopAlg; ElectAlg; TravAlg
                           Op
                           }
```

From the implementation we only show a few methods referred to in later code snippets:

```
const inf = uint(1 << 16)
var (done = make(chan int, 1); p0 = nchan.Port0())

func value (a Any) uint { return a.(vtx.Vertex).Content().(Valuator).Val() }

func new_(g gra.Graph) DistributedGraph {
  x := new(distributedGraph)
  x.Graph = g
  x.bool = x.Graph.Directed()
  x.n = x.Graph.Num() - 1
  x.actVertex = x.Graph.Get().(vtx.Vertex)
  x.me = value(x.actVertex)
  if x.me != ego.Me() { ker.Panic("chaos: me != Me !") }
  x.tmpVertex = vtx.New (x.actVertex.Content(), x.actVertex.Wd(), x.actVertex.Ht())
  v0 := g.Neighbour(0).(vtx.Vertex)
  g.Ex2 (x.actVertex, v0)
  if g.Edged() {
    x.tmpEdge = g.Get1().(edg.Edge)
  } else {
    g.Ex2 (v0, x.actVertex)
    x.tmpEdge = g.Get1().(edg.Edge)
  }
  x.Graph.SetWrite (vtx.W, edg.W)
  x.tmpGraph = Clone(x.Graph).(gra.Graph)
  x.tree = gra.New (true, x.tmpVertex, x.tmpEdge); x.tree.SetWrite (x.Graph.Writes())
```

```
  x.ring = gra.New (true, x.tmpVertex, x.tmpEdge); x.ring.SetWrite (x.Graph.Writes())
  x.visited = make([]bool, x.n)
  x.nb, x.nr = make([]vtx.Vertex, x.n), make([]uint, x.n)
  x.ch = make([]nchan.NetChannel, x.n)
  x.parent, x.child = inf, make([]bool, x.n)
  x.actHost = host.Localhost()
  x.host, x.port = make([]host.Host, x.n), make([]uint16, x.n)
  x.sport, x.cport = make([]uint16, x.n), make([]uint16, x.n)
  x.mon = make([]fmon.FarMonitor, x.n)
  x.monM = make([]fmon.FarMonitorM, x.n)
  for i := uint(0); i < x.n; i++ {
    g.Ex (x.actVertex)
    x.nb[i] = g.Neighbour(i).(vtx.Vertex)
    x.nr[i] = x.nb[i].(Valuator).Val()
    x.Graph.Ex2 (x.actVertex, x.nb[i])
    v := x.Graph.Get1().(edg.Edge).(Valuator).Val()
    ps := v / (1<<20)
    pc := (v - 1<<20 * ps) / (1<<10)
    x.port[i] = nchan.Port0 + uint16(v - 1<<10 * pc - 1<<20 * ps)
    x.sport[i], x.cport[i] = nchan.Port0 + uint16(ps), nchan.Port0 + uint16(pc)
    if x.nr[i] < x.me { x.sport[i], x.cport[i] = x.cport[i], x.sport[i] }
  }
  g.Ex (x.actVertex)
  x.rank = uint(16) // for graphs with up to 16 vertices
  x.top = adj.New (x.rank, uint(1))
  for i := uint(0); i < x.n; i++ {
    x.top.Set (x.me, x.nr[i], uint(1)); x.top.Set (x.nr[i], x.me, uint(1))
  }
  x.TopAlg, x.ElectAlg, x.TravAlg = PassGraph, ChangRoberts, DFS
  x.leader = x.me
  x.Op = Ignore
  return x
}

func (x *distributedGraph) SetHosts (h []host.Host) {
  if uint(len(h)) != x.n { ker.Shit() }
  for i := uint(0); i < x.n; i++ { x.host[i] = h[i] }
}

func (x *distributedGraph) connect (a Any) {
  for i := uint(0); i < x.n; i++ {
    x.ch[i] = nchan.New (a, x.me, x.nr[i], x.host[i], x.port[i])
  }
}

func (x *distributedGraph) fin() {
  for i := uint(0); i < x.n; i++ { x.ch[i].Fin() }
}

func (x *distributedGraph) finMon() {
  for i := uint(0); i < x.n; i++ { x.mon[i].Fin() }
}

func (x *distributedGraph) finMonM() {
  for i := uint(0); i < x.n; i++ { x.monM[i].Fin() }
}

func (x *distributedGraph) channel (id uint) uint {
  j := x.n
  for i := uint(0); i < x.n; i++ { if x.nr[i] == id { j = i break } }
  return j
}
```

```
func (x *distributedGraph) decodedGraph (bs []byte) gra.Graph {
  g := gra.New (x.tmpGraph.Directed(), x.tmpVertex, x.tmpEdge)
  g.Decode(bs)
  g.SetWrite (vtx.W, edg.W)
  return g
}

func (x *distributedGraph) directedEdge (v, v1 vtx.Vertex) edg.Edge {
  x.tmpGraph.Ex2 (v, v1)
  e := x.tmpGraph.Get1().(edg.Edge)
  e.Direct (true)
  e.SetPos0 (v.Pos()); e.SetPos1 (v1.Pos());
  e.Label (false)
  return e
}

func nrLocal (g gra.Graph) uint { return value (g.Get()) }

func valueMax (g gra.Graph) uint {
  m := uint(0); g.Trav (func (a Any) { v := value (a.(vtx.Vertex)); if v > m { m = v } })
  return m
}

func exValue (g gra.Graph, v uint) bool {
  return g.ExPred (func (a Any) bool { return a.(vtx.Vertex).Val() == v })
}
```

For the used packages on graphs, vertices and edges and vertices (`gra`, `vtx` and `edg`) we again refer to murus.
So a user of the package `dgra` may construct arbitrary graphs—directed or not—as examples to study effects of distributed algorithms.

Topology of a network

As first example of a distributed algorithem we show the algorithm to compute the topology of a local network (see e.g. [A p. 373], [A1 p. 450] or [R p. 7]).
A conceptual difference to the solution of the left right problem with a far monitor is, that in this case—as in all other later examples of distributed algorithms—messages are not only passed from every client to the (central) server and returned from it, but also between clients. So each client has to play the role of a server to be able to provide monitor operations to his neighbours.
As noted in the previous section, at the beginning every vertex knows only his "star", i.e. the processes residing on hosts being connected by communication lines with the host on which the current process is working.
The idea is, that each process starts the algorithm by sending the information about his neighbourhood to all of its neighbours and gets back the corresponding knowledge from each of them. This step is iterated, so in each round the graph grows until it contains all vertices and edges of the global graph.
In [A] the passed information is coded in form of an adjacency matrix with rank = global (!) number of participants.
Instead of sending matrices of a fixed rank around, we prefer to pass the growing graphs themselves (similar as in [R], coded as byte streams of variable length (the mentioned package `murus/gra` provides the necessary methods). The solution can easily be implemented with the help of far monitors:

```
package dgra

import (. "murus/obj"; "murus/fmon")

func (x *distributedGraph) fmgraph() {
  go func() {
    fmon.NewM (nil, 1, x.n, x.nr, x.addG, AllTrueSp, x.actHost, x.sport, true)
  }()
```

```
  for i := uint(0); i < x.n; i++ {
    x.monM[i] = fmon.NewM (nil, 1, i, x.nr, x.addG, AllTrueSp, x.host[i], x.cport, false)
  }
  defer x.finMonM()
  x.awaitAllMonitorsM()
  x.tmpGraph.Copy (x.Graph)
  x.tmpGraph.Ex (x.actVertex)
  x.tmpGraph.SubLocal()
  x.tmpGraph.Write()
  lock <- 0
  for r := uint(1); r <= x.diameter; r++ {
    for i := uint(0); i < x.n; i++ {
      <-lock
      bs := x.tmpGraph.Encode()
      lock <- 0
      bs = x.monM[i].Fm (bs, 0, i).([]byte)
      g := x.decodedGraph(bs)
      <-lock
      x.tmpGraph.Add (g)
      x.tmpGraph.Ex2(x.actVertex, x.nb[i])
      x.tmpGraph.Sub2()
      x.tmpGraph.Write()
      lock <- 0
    }
  }
}

func (x *distributedGraph) addG (a Any, i uint) Any {
  x.awaitAllMonitorsM()
  <-lock
  x.tmpGraph.Add (x.decodedGraph(a.([]byte)))
  bs := x.tmpGraph.Encode()
  lock <- 0
  return bs
}
```

The function `AllTrueSp` denotes a trivial predicate spectrum, defined in the package `obj`:

```
func AllTrueSp (a Any, i uint) bool { return true }
```

The function call `x.tmpGraph.Write()` writes an image of the graph to the screen, which gives a nice visualization of the algorithm.
For reasons not to be discussed here, we used multi far monitors, that provide different channels for each client-server connection. Their specification is:

```
type
  FarMonitorM interface {
  Fm (a Any, i, k uint) Any
  Fin()
}

// Pre: j is the id of the calling process,
//      nr are the ids of the neighbours of the calling process.
// See above. Additionally, for each server-client-pair there is an own channel.
func NewM (a Any, n, j uint, nr []uint, fs FuncSpectrum, ps PredSpectrum,
           h host.Host, p []uint16, s bool) FarMonitorM {
  return newM (a, n, j, nr, fs, ps, h, p, s)
}
```

Now, this solution is not consistent with our postulated restriction, because it uses the diameter of the global graph.
But with a bit more effort this can be remedied using the idea of [A p. 375]. We simply mark subgraphs, starting with the calling vertex and respect them while adding the graph. Then the termination is given by testing, if the subgraph is the whole graph:

```
func (x *distributedGraph) fmgraph1() {
  ... same as above
  known := false
  active := make([]bool, x.n)
  for i := uint(0); i < x.n; i++ { active[i] = true }
  r := uint(1)
  for ! known {
    for i := uint(0); i < x.n; i++ {
      <-lock
      bs := x.tmpGraph.Encode()
      lock <- 0
      bs = x.monM[i].Fm (append(Encode(false), bs...), 0, i).([]byte)
      g := x.decodedGraph(bs[1:])
      <-lock
      if Decode(false, bs[:1]).(bool) { active[i] = false }
      x.tmpGraph.Add (g)
      x.tmpGraph.Ex2(x.actVertex, x.nb[i])
      x.tmpGraph.Sub2()
      x.tmpGraph.Write()
      lock <- 0
    }
    if ! known && x.tmpGraph.EqSub() { known = true }
    r++
  }
  for i := uint(0); i < x.n; i++ {
    if active[i] {
      x.monM[i].Fm (append(Encode(false), x.tmpGraph.Encode()...), 0, i)
    }
  }
}

func (x *distributedGraph) addG1 (a Any, i uint) Any {
  x.awaitAllMonitorsM()
  <-lock
  x.tmpGraph.Add (x.decodedGraph(a.([]byte)[1:]))
  bs := append(Encode(true), x.tmpGraph.Encode()...)
  lock <- 0
  return bs
}
```

Network traversal by depth-first-search

One of the most cited DFS-traversals in an network is the algorithm of Awerbuch in [Aw].
This can be simplified with far monitors: If you compare the original with the following version, you might find, that our solution is indeed quite elegant and much easier to understand than the original.

```
func (x *distributedGraph) fmdfsa (o Op) {
  ... same as in other algorithms
  if x.me == x.root {
    x.parent = x.me
    for k := uint(0); k < x.n; k++ { x.mon[k].F(x.me, 0) }
    for k := uint(0); k < x.n; k++ {
      if ! x.visited[k] {
        x.visited[k], x.child[k] = true, true
        x.mon[k].F(x.me, 1)
      }
    }
    x.Op(x.actVertex)
  } else {
    <-done
  }
}
```

```
func (x *distributedGraph) da (a Any, i uint) Any {
  x.awaitAllMonitors()
  s := a.(uint)
  j := x.channel(s)
  switch i {
  case 0:
    x.visited[j] = true
  case 1:
    x.parent = x.nr[j]
    for k := uint(0); k < x.n; k++ { if k != j { x.mon[k].F(x.me, 0) } }
    for k := uint(0); k < x.n; k++ {
      if k != j && ! x.visited[k] {
        x.visited[k], x.child[k] = true, true
        x.mon[k].F(x.me, 1)
      }
    }
    x.Op(x.actVertex)
    done <- 0
  }
  return x.me
}
```

The resulting depth-first-search-tree is given by the values of the fields `x.parent` and `x.child[x.me]` for every vertex.
Awerbuch's "trick" to use the visit phase plays a necessary role in our algorithm, because otherwise the far monitor calls would run into a deadlock: If we try to do the job just in one phase a vertex blocked after a call of a monitor operation would get stuck when the called monitor catches a call from another vertex before he returns his result–a classical desaster with "nested monitors" (see e.g. [L]).
It is easy to see that along every edge exactly two pairs of messages are passed ("pair" means the call of a monitor function and the return of its answer): Either one pair in a visit and one in a discover phase (both in the same direction) or two pairs in a visit phase in opposite directions. So the message complexity is exactle $4 \times |E|$ (E = number of egdes in the graph).
It should be noted that this algorithm can also be generalized by message transport of the evolving trees instead just the identities of the processes to give also a nice graphical output.

Construction of a unidirectional ring

As the classical algorithms for the "election of a leader" in a network (e.g. [CR], [P], [DKR] and [HS]) are restricted to networks consisting of a uni- or bidirectional ring, it is plausible to construct such a ring from an arbitrary network graph.
Such a construction is possible by using the idea of Awerbuchs algorithm presented in the previous section.
The additional idea is for every vertex to protocol the visit time in the field `x.time0` by increasing the received time and to pass this value in the call to the next vertex. So the sequence of the vertices in the ring is given by the discover-times time:

```
func (x *distributedGraph) fmdfsring() {
  ... same as in other algorithms
  if x.me == x.root {
    for k := uint(0); k < x.n; k++ { x.mon[k].F(x.me, 0) }
    x.time = 0
    for k := uint(0); k < x.n; k++ {
      if ! x.visited[k] {
        x.visited[k] = true; x.mon[k].F(x.me + x.time * inf, 1)
      }
    }
  } else {
    <-done
  }
}
```

```
func (x *distributedGraph) dr (a Any, i uint) Any {
  x.awaitAllMonitors()
  s := a.(uint) % inf; j := x.channel(s)
  switch i {
  case 0:
    x.visited[j] = true
  case 1:
    for k := uint(0); k < x.n; k++ { if k != j { x.mon[k].F(x.me, 0) } }
    t := a.(uint) / inf; t++; x.time = t
    for k := uint(0); k < x.n; k++ {
      if k != j && ! x.visited[k] {
        x.visited[k] = true; t = x.mon[k].F(x.me + t * inf, 1).(uint) / inf
      }
    }
    done <- 0
    return x.me + inf * t
  }
  return 0
}
```

Of course also this algorithm can be generalized towards graphical output on the screen.

Election of a leader

Apart from the algorithms mentioned in the previous section there are indeed some algorithms for the election of a leader in graphs, that are not simply rings, e.g. in [KMZ]. But–as Peterson remarks in [P1]–this "algorithm is extremely long and complicated and it is very difficult to determine its corectness". A leader can be found much easier and without any restriction on the underlying graph: simply by depth-first-search.
The previously presented dfs-algorithms have just to be enriched by passing the current maximum of the identities of the processes involved: Every process compares its identity with the received value and returns the maximum of both. Of course finally a third phase is needed to propagate the value to all processes:

```
func (x *distributedGraph) fmdfse() {
  ... same as in other algorithms
  if x.me == x.root {
    x.parent = x.me
    for k := uint(0); k < x.n; k++ { x.mon[k].F (x.me + inf * x.leader, 0) }
    for k := uint(0); k < x.n; k++ {
      if ! x.visited[k] {
        x.visited[k], x.child[k] = true, true
        v := x.mon[k].F (x.me + inf * x.leader, 1).(uint); if v > x.leader { x.leader = v }
      }
    }
    for k := uint(0); k < x.n; k++ { if x.child[k] { x.mon[k].F(x.me + inf * x.leader, 2) } }
  } else {
    <-done
  }
}

func (x *distributedGraph) de (a Any, i uint) Any {
  x.awaitAllMonitors()
  s, v := a.(uint) % inf, a.(uint) / inf
  j := x.channel(s)
  switch i {
  case 0:
    x.visited[j] = true
  case 1:
    x.parent = s
    if v > x.me { x.leader = v }
    for k := uint(0); k < x.n; k++ { if k != j { x.mon[k].F(x.me + inf * x.leader, 0) } }
```

```
    for k := uint(0); k < x.n; k++ {
      if k != j && ! x.visited[k] {
        x.visited[k], x.child[k] = true, true
        v = x.mon[k].F(x.me + inf * x.leader, 1).(uint)
        if v > x.leader { x.leader = v }
      }
    }
    return x.leader
  case 2:
    x.leader = v
    for k := uint(0); k < x.n; k++ { if x.child[k] { x.mon[k].F(x.me + inf * x.leader, 2) } }
    done <- 0
  }
  return x.me
}
```

Network traversal by breadth-first-search

Breadth-first-search is a well known technique, used e.g. to solve problems like the search for a "shortest path" between two vertices. Clearly also this type of net traversal can be handled with far monitors.
Root starts in a loop with the operating distance 0 and calls the first monitor operation of his neighbours with his identity and this distance.
A called monitor marks the caller as visited. As first call it receives the operation distance `r == 0`. In this case, if its parent is already defined (less than `inf`), it returns `inf` to inform the caller about having another vertex as parent; otherwise it marks the caller as its parent and returns its id (less than `inf`).
In a later call (iff `r > 0`) it executes monitor calls to all its neighbours with the operation distance `r` decreased by 1. If it gets back the value `inf`, it marks that neighbour as visited, otherwise as one of its children. It returns its identity, iff there are more neighbours to explore, otherwise again `inf` to inform the caller, that it is pointless to send any more monitor calls.
Root does the same:
If in its loop gets it back `inf` from a monitor call, it knows that that neighbour has already another parent and marks it as visited, otherwise as one of its children. If there are not any more processes returning an identity less than `inf`, the loop is left and the search has terminated.
So it is left to root, to start the propagation of the breadth-first-tree to all its children, who do the same. So finally every process has the breadth-first-tree.

```
func (x *distributedGraph) fmbfs (o Op) {
  ... same as in the previous algorithm
  if x.me == x.root {
    x.parent = x.me
    for {
      c := uint(0)
      for k := uint(0); k < x.n; k++ {
        if ! x.visited[k] {
          if x.mon[k].F(x.me + inf * x.distance, 0).(uint) == inf {
            x.visited[k] = true
          } else {
            x.child[k] = true; c++
          }
        }
      }
      if c == 0 { break }
      x.distance++
    }
    for k := uint(0); k < x.n; k++ { if x.child[k] { x.mon[k].F(x.me, 1) } }
    x.Op (x.actVertex)
  } else {
    <-done // wait until root finished
  }
}
```

```
func (x *distributedGraph) b (a Any, i uint) Any {
  x.awaitAllMonitors()
  s := a.(uint) % inf; j := x.channel(s)
  x.distance = a.(uint) / inf
  switch i {
  case 0:
    x.visited[j] = true
    if x.distance == 0 {
      if x.parent < inf { return inf }
      x.parent = s // == x.nr[j]
      x.Op (x.actVertex)
      return x.me
    }
    c := uint(0) // r > 0
    for k := uint(0); k < x.n; k++ {
      if k != j && ! x.visited[k] {
        if x.mon[k].F(x.me + (x.distance - 1) * inf, 0).(uint) == inf {
          x.visited[k] = true
        } else {
          x.child[k] = true
          c++
        }
      }
    }
    if c == 0 { return inf }
  case 1:
    for k := uint(0); k < x.n; k++ {
      if x.child[k] { x.mon[k].F(x.me, 1) }
    }
    done <- 0
    return inf
  }
  return x.me
}
```

The use of the operation distance with its increments and decrements could be considered as a sort of virtual recursion, replacing the real recursion in the classical sequential breadth-first-search algorithm. Of couse also for this algorithm murus provides a nice graphical output to the screen.

References

[A] Andrews, G. R., *Concurrent Programming, Principles and Practice*, Addison-Wesley, 1991.

[A1] Andrews, G. R., *Foundations of Multithreaded, Parallel and Distributed Programming*, Addison-Wesley, 2000.

[Aw] Awerbuch, B., *A New Distributed Depth-First-Search Algorithm*, Inf. Proc. Letters **20** (1985), 147–150.

[BD] Burns, A., Davies, G., *Concurrent Programming*, Addison-Wesley, 1993.

[CR] Chang, E., Roberts, R., *An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes*, Commun. ACM **22** (1979), 281–283.

[DKR] Dolev, D., Klawe, M., Rodeh, M., *An O(nlogn Unidirectional Distributed Algorithm for Extrema Finding in a Circle*, Journal of Algorithms **3** (1982), 245–260.

[H] Hoare, C. A. R., *Monitors: An Operating Systems Structuring Concept*, Commun. ACM **17** (1974), 549–557.

[H1] Hoare, C. A. R., *Communicating Sequential Processes*, Commun. ACM **21** (1978), 666–677.

[HS] Hirschberg, D. S.., Sinclair, J. B., *Decentralized Extrema Finding in Circular Configurations of Processes*, Commun. ACM **23** (1980), 627–628.

[KMZ] Korach, E., Moran, S., Zaks, S., *Tight Lower and Upper Bounds for Some Distributed Algorithms for a Complete Network of Processors*, TAP, ACM Conference (1984), 199–207.

[L] Lister, A., *The Problem of Nested Monitor Calls*, ACM SIGOPS Oper. Syst. Rev. **11** (1977).

[P] Peterson, G. L., *An nlogn Unidirectional Algorithm for the Circular Extrema Finding Problem*, ACM Trans. Program. Lang. Syst. **4** (1982), 758–762.

[P1] Peterson, G. L., *Efficient Algorithms for Elections in Meshes and Complete Graphs*, TR 140, Dept. of Comp. Sci., University of Rochester (1985).

[R] Raynal, M., *Distributed Algorithms for Message-Passing Systems*, Springer, 2013.

Institut für Informatik, Freie Universität Berlin, Takustr. 9, 14195 Berlin

E-mail address: christian.maurer@fu-berlin.de

URL: www.inf.fu-berlin.de/users/maurer