Rumor-spreading Simulation using Parallel Scale Free Networks

Pushkar Godbole, Parminder Bhatia, Kaushik Patnaik

April 27, 2015

1 Introduction

Scale-free networks are graphical networks whose degree distribution follows a power law. The recent interest in scale free networks has been spurred mainly by the observance of close similarity between real-world networks and their scale free models. In particular, the topologies of the world-wide-web and many other social and biological communication networks have been observed to approximately follow the power law. All these instances essentially have an information diffusion model at their core. Among its myriad applications, the one of particular interest to us is the spread of rumors through social networks. A simulation based analysis of this phenomenon becomes particularly interesting in the present setting of large scale online information dissemination through social networks.

2 Objective

The aim of this project is to develop an queuing network based rumor spreading simulation using a parallel scale free network implementation. The objectives can be broadly classified into two categories:

- **Serial vs Parallel:** It has been observed that, as the size of the network increases, sequential simulations increasingly exceed tolerable limits w.r.t. simulation runtimes and memory consumption [2]. Hence the aim of this project has been to develop a parallel implementation of the scale free network simulation. The parallel implementation has been evaluated for scale-up and accuracy as the number of processors increases by comparing it to serial execution.
- **Spread Policy:** The second part of the project is the evaluation of various rumor spreading policies at each node. These policies are evaluated for the range of spread and robustness of the information dissemination protocol, subject to the given network topology.

Figure 1: Scale Free Networks [1]

3 Literature Survey

In this section we describe the previous and relevant work done in the domain of Parallel Scale Free Networks. Garbinato et al [3] show that scale-free topologies have a positive impact on the performance of gossip-algorithms in peer to peer networks. They show that in such topologies, gossip algorithms tend to require fewer number of messages and exprience smaller latency compared to other topologies such as rings or grids. Gabriele D'Angelo et al [4] presented a new simulation tool for scale-free netorks with high number of nodes. Their tool enables parallel and distribtuted simulation of scale-free networks. To demonstrate efficiency, they analysed performance of a few Gossip Algorithms. Although, they were not able to achieve large speedup over serial run-times, they layed the foundation for Parallel Scale free Networks. Pienta et al [2] develop an analytical model to study parallelism available in simulations of scale-free newtorks. They study the performance of such simulations using two-variants of Chandy/Mishra/Bryant synchronization algorithms, and highlight the importance of topology in such simulations.

Apart from the conceptual model, our implementation is mainly based on three external packages. The Networkx package [5] for Python is used for the synthetic graph generation. With the network generated, the graph needs to be distributed across processors for executing the parallel implementation. The efficient partitioning of the graph is handled using PyMetis (Python wrapper for Metis) [6]. Finally, the parallel scale free queuing network simulation model is built using mpi4py [7] Initially, the Python package simx [8] was used for handling the parallelization. However, scalability issues with simx led to adoption of a first principles approach using mpi4py.

4 Model

The core of the simulator is a distributed queuing network model based on the graph topology. The aim of the implementation has been to develop an event generation and execution model to be executed in parallel relying on a synchronization algorithm to interface across processors. These tasks remain broadly similar to the Logical Process (LP) based parallel simulation approach described in class.

4.1 Queuing Network Simulator

The basic model for simulating the information dissemination through the network is a Parallel Discrete Event Simulation (PDES) synchronously simulating events over multiple processors. The PDES program consists of a collection of logical processes (LPs) each representing a parallel processor that communicates by exchanging timestamped messages.

A message may be generated at any node of the scale free network during initialization. The simulator ensures the dissemination of information from every source node based on the defined gossip protocol. The events associated with each node essentially are:

- Send message (to a neighbor)
- Receive message (from a neighbor)

The time delay between the send-receive pair is equal to the length of the edge between the corresponding nodes. We assume that the graph does not have non-zero or self edges. This ensures that the simulation wouldn't enter a livelock at any point of time.

A synchronization mechanism is necessary to ensure appropriate exchange of events across processors in the parallel execution, so that it produces exactly the same results as the corresponding sequential execution. The synchronization approach used for maintaining concurrency across all LPs is conservative and uses the the minimum message traversal time across each active edge over all processors for time-stepping.

Each processor serially executes the message passsing tasks in its event queue, based on their time-stamps. At the end of each global time-step, each processor determines the shortest edge length over all internal active nodes. This serves as the local Lower Bound Time Stamp (LBTS) for that processor. All processors then broadcast and reduce their local LBTS values based on

the global minimum. This global minimum serves as the time-step for the subsequent execution.

At the end of every global time-step, all processors determine the outbound messages, their corresponding nodes and destination processors. These messages are then scattered by all processors to their respective destinations before switching back to the subsequent execution.

4.2 Gossip Algorithms

In this section we describe the algorithms we consider to gossip messages on scale free networks.

In our model, all nodes or some subset or nodes are able to generate the a new message. When the message generation is invoked in a node, a single message my be created with a certain probability. The message can be a string, or a value. Each message has a limited time of survival, and can only be propagated within this ttl (time to live).In our model the message ttl is set at a sufficiently high value compared to the LBTS value, to enable messages to propagate certain distances from the origin nodes. Each message has a unique identifier associated with it. A node is allowed to gossip out a particular message only once. This avoids formation of message cycles and prevents messages from being trapped in a loop. A node ignores the reception of a message that was previously transmitted by it. Once the message is created, it is propagated through the net using the Gossip algorithms described below.

Upon receiving a message, the receiving node first updates the ttl value of the message and then forwards the message to its own neighbors. The receive procedure ensures the message is not sent back to node the message came from. Unlike [4] who cache the messages to avoid repeat of messages, we believe this is a more natural message passing model.

We consider four different gossip algorithms, which are partly inspired from [4]. All algorithms require knowledge of the message and graph structure to perform gossip.

4.2.1 Fixed Probability Algorithm

Data: Graph Nodes, Message, and Neighborhood Structure
initialization;
while *iterating over neighbors* **do**
  if *random < 0.5* **then**
    sendMessage(msg, neighbor);
  else
    dont send message
  end
end

Algorithm 1: Fixed Probability Algorithm

In this algorithm, each node has asymptotically a 50-50 chance to send the message to one of its neighbors. For each of its neighbors a coin-flip decides the propagation of the message. Thus nodes with higher number of neighbors will send a higher number of messages, compared to other nodes. Thus the computation work performed at these nodes is higher.

4.2.2 Distance Probability Algorithm

Data: Graph Nodes, Distance, Message, and Neighborhood Structure
initialization;
while *iterating over neighbors* **do**
  if *random < 1/distance to neighbor* **then**
    sendMessage(msg, neighbor);
  else
    dont send message
  end
end

Algorithm 2: Distance Probability Algorithm

As a modification of the above algorithm we also consider the distances to the node's neighbors as a variable affecting the spread of gossip. The farther the neighbor is, the lesser the probability of propagating the message.The distance metric can also be replaced by weight, which signifies the level of similarity or closeness between two nodes.

4.2.3 Fixed Fanout Probability Algorithm

Data: Graph Nodes, Message, Fanout(k) and Neighborhood Structure
initialization;
calculate rng = min(k, length(neighbors);
while *iterating over rng* **do**
  randomly pick a neighbor sendMessage(msg, neighbor);
end

Algorithm 3: Fixed Fanout Probability Algorithm

Instead of propagating messages probabilistically, we can instead threshold the total number of messages that a node can receive and send in a single iteration(called a fanout). If fanout =1, then each node randomly selects one neighbor to transmit the message to. If fanout is greater than the number of neighbors, then all the neighbors are sent the message.

4.2.4 Variable Probability Algorithm

Data: Graph Nodes, Message, and Neighborhood Structure
initialization;
calculate thres = length(neighbors)/max(neighbors over all nodes);
while *iterating over neighbors* **do**
  if *random* < *thres* **then**
    sendMessage(msg, neighbor);
  else
    dont send message
  end
end

Algorithm 4: Variable Probability Algorithm

Data: Graph Nodes, Message, and Neighborhood Structure
initialization;
calculate thres = length(neighbors)/max(neighbors over all nodes);
while *iterating over neighbors* **do**
  if *random* < *thres* **then**
    sendMessage(msg, neighbor);
  else
    dont send message
  end
end

Algorithm 5: Variable Probability Algorithm

As a slight modification of the 1st gossip algorithm, instead of 50-50 we specify higher propagation chances to nodes with large number of neighbors. More specifically, the chance of propagation is directly proportional to number of neighbors a node has. This specification reduces communication in most of the nodes, with the hub like nodes being the center of all communication. If the graph is partitioned so as to keep all hub-like nodes in a single lp, the communication between lps will be minimal resulting in massive speedup.

5 Approach

The implementation of the rumor spreading model can be broadly divided into three parts:

- Graph Generation
- Graph Partitioning
- Gossip simulation

Here we describe the above three steps and describe their implementation in detail.

5.1 Graph Generation

Although recently many real world networks have been shown to exhibit scale-free properties, they may have complex topologies making them difficult to evaluate. Additionally, real-world graphs lack customizability that would be preferable for the testing and evaluation of our algorithms. Hence we use both synthetically generated scale-free graphs and scale-free-like real world networks for running the simulation.

For the generation of the synthetic scale-free graphs we use the Python based **Networkx** library. Networkx enables generation of scale free graphs, given various input parameters such as graph-size, α, β, γ probabilities, etc.

After evaluating the simulation on the synthetic graph, we would run the same on real-world networks. [9]. The primary source of real-world graphs would be the openly available social-network data. We plan to use the graph datasets available on Stanford Network Analysis Project (SNAP) [10] for the same.

5.2 Graph Partitioning

Running the scale free rumor spreading simulation in parallel implies distribution the graph data across processors. This partitioning of the graph into subgraphs to be distributed amongst participating processors needs to be such that, it ensures:

- **Load-balancing:** Near-equal distribution of nodes such that all processors handle approximately equal workload.
- **Minimal Data-exchange:** Minimum surface area between processor sub-graphs that ensures minimal data exchange across processors.

PyMetis, the python wrapper for Metis [6] has been used for partitioning the graph. Metis ensures that it partitions the graph along the min-cut such that the generated sub-graphs are approximately equal.

5.3 Gossip simulation

Finally with the graph data generated and partitioned, the final step is to execute the rumor spreading simulation and evaluate the results. The simulation is based on a queuing network approach where each processor holding its own sub-graph acts as a Logical Process (LP) and interface with other processors for exchange of data across the boundaries. It may be noted that, although each processor only handles the gossips on its assigned subgraph, it has the entire graph topology with the node to processor mapping, as delivered by the graph-partitioner. This ensures that all processors know the destinations of their outgoig messages. The two key aspects necessary for accurate simulation w.r.t. the serial simulation are:

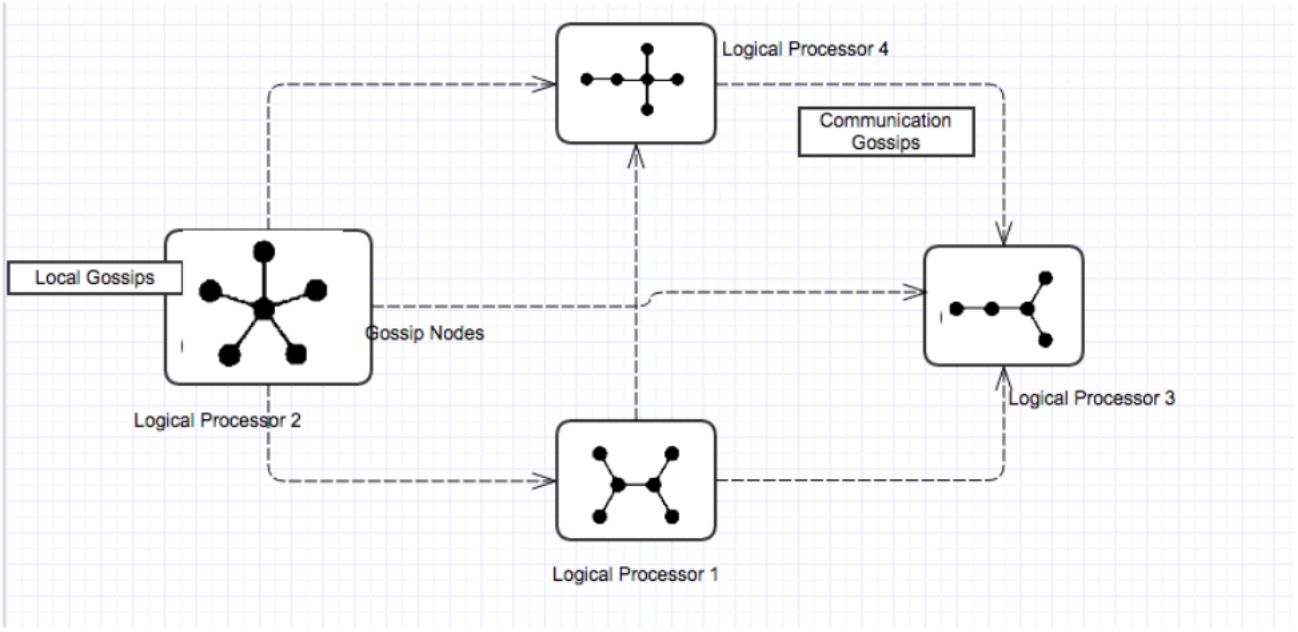

Figure 2: Architecture for Communication between LP's

- LP Synchronization
- Data Exchange

5.3.1 Initial Approach Using SimX

Initially, the Python based **SimX** library was used for modeling and executing the simulation. SimX is a generic library for developing parallel, discrete event simulations in Python, designed as a skeleton for data distribution and parallelization for discrete event simulations. SimX has an inbuilt framework to handle MPI calls and synchronization. However, Simx has no option to choose custom data distribution and synchronization strategies. Additionally, Simx failed at scaling up to order million nodes. This necessitated the implementation of a generic scalable parallelization framework for our simulator. Hence we chose to completely re-implement the parallel simulator using mpi4py.

5.3.2 MPI Based Approach

Implementing the MPI based communication from scratch, has enabled us to choose our data partitioning strategy and optimize for scale-up in large datasets.

As can we observed from the pseudo code from the algorithm, we used **Process oriented approach**, where we execution followed flow for a Node. Each LP consisted of cluster of Nodes obtained from the Graph Partitioning Algorithm.

There are basically two types of gossips:

- **Local Gossips** - These are the gossips where source and destination are there on the same LP.
- **Communication Gossip** - These are the gossips where source and destination are there on the same LP.

Gossips are sent after the global time stamp which is the min of length of all active edges across the graph thus ensuring events are synchronized.
Each of the nodes maintain the following data structure required for Inter Process Communication.

GossipNode
Gossip node has the following attributes.

- id - unique id for Node.
- processorid - Processor /LP to which the Node belongs
- neighbours(id,weight) - id and edge weight for the neighbours.
- sentgossips - List of gossips already sent from this Node. This ensures that no node sends the same gossip twice to same neighbour.
- algorithm - Algorithm associated for Gossips.

Gossips
Gossip has the following attributes.

- id - unique id for Gossip.
- data/intensity - Data or intensity of the Gossip. This value may decay as it goes further in the network.
- time- Time when is gossip is to be executed / forwarded to neighbour
- ttlalgo - Algorithm on how the intensity might decay as the gossip traverses the graph.
- source -Source for Gossips.

Time Stamp is incremented through Broadcast from Root Processor.
After each Time Stamp **Communication Gossip** are exchanged between the processors . For MPI we basically use **AllScatter** to scatter boundary messages from each processor to their appropriate destinations.

```
1:  procedure ALGORITHM
2:      while currtime > endtime do
3:          for node in nodes do
4:              if Gos in LocGos[node.id] then
5:                  sendGossip())
6:              end if
7:          end for
8:          for i in processors do
9:              if i is not rank then
10:                 scatter(ComGos[i]))
11:             end if
12:         end for
13:         if rank is 0 then
14:             broadcast(currtime+dt))
15:         end if
16:     end while
17: end procedure
```

Figure 3: Parallel Algorithm per processor

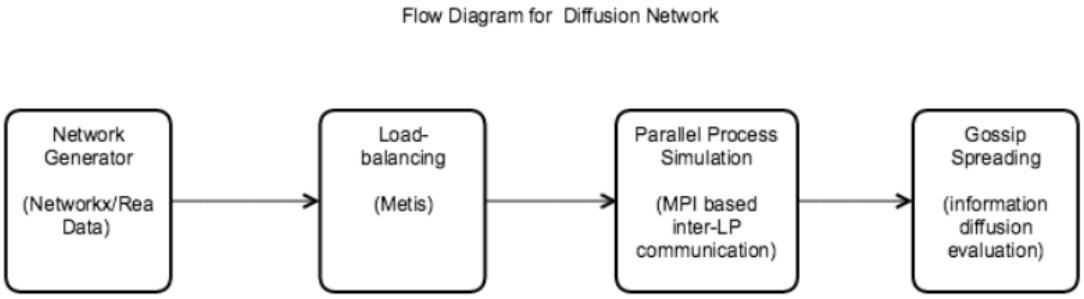

Figure 4: Process Flow Diagram

6 Results

We tested our simulations on two real world social network graphs: Twitter and Facebook . The Twitter dataset contain nodes and edges, and the Facebook dataset contains nodes and edges. Social networks have been found to exhibit scale-free behavior with few nodes of large degree and a long tail of low degree nodes. The Jinx cluster has been used for the implementation and execution of the simulation.

To evaluate the difference between different gossip algorithms, we evaluated the total number of gossips that get transmitted in the simulation per time. As we observe in Figure 5, the number of gossips transmitted if highest for the HighDegree gossip algorithm and lowest for FixedFan type gossip algorithm. However, the gossips transmitted by HighDegree algorithm exponentially decrease with time, which is expected due to scale free network nature of the dataset. FixedProb and DistProb algorithms show similar behavior (exponential decrease) although with a much larger variance across time.

To evaluate the traversal of gossips with changing parameters, we ran the FixedProb algorithm with different probability thresholds and plotted average number of nodes visited by a gossip. Since gossips from a single source can take several different paths, we sum across all such paths to count the number of nodes visited by a gossip. We then average across all such gossips to obtain the average number of nodes visited. As we can observe in Figure 6, with increasing probability thresholds there is increase in average number of nodes visited.

Finally to evaluate the runtime of our simulations, we ran the FixedProb gossip algorithm on Facebook dataset with different number of processors. As we can observe in Figure 7, we do see a speed using parallel simulations with runtime decreasing from 110s to 15s with 8 processors. However, this behavior is not asymptotic as the communication cost increases eventually to dominate the runtime. This can be observed by the increase in runtime to 40s with 16 processors.

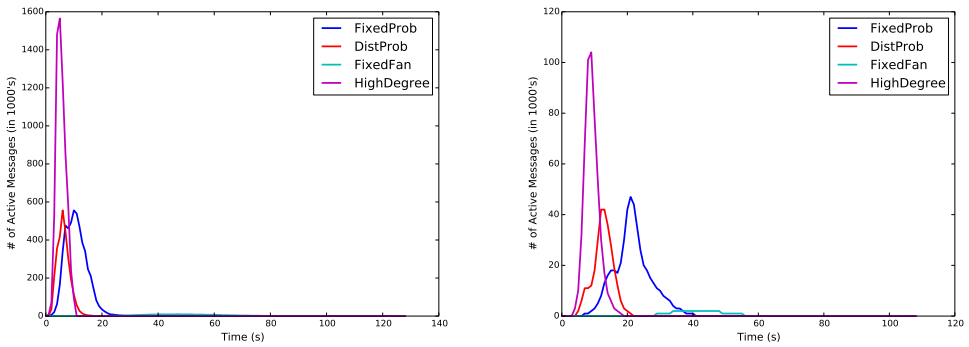

Figure 5: Total number of gossips transmitted in simulation with time

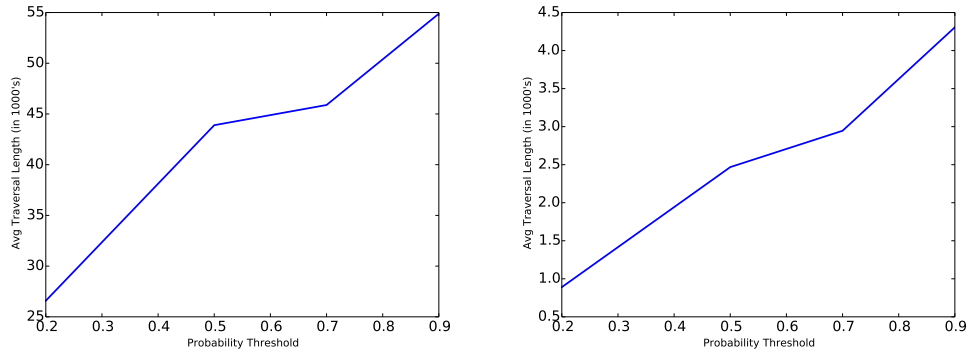

Figure 6: Avg Number of Nodes visited by a Gossip with different transmission probabilities for FixedProb Gossip

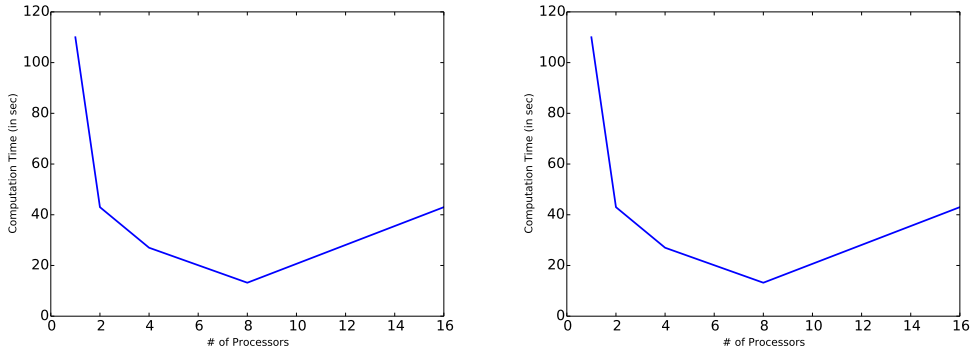

Figure 7: Computation Time with FixedProb gossip algorithm for different number of processors

7 Conclusions & Future Work

We implemented and tested a scale-free network simulator using Python-MPI. The scalability of our implementation permits the simulation of networks composed by a high number of nodes. With a simple synchronization scheme we are able to demonstrate speedup over serial implementations. We also evaluated several different Gossip algorithms and their performance in real-world graphs.

To extend this work further, more modern gossip algorithms developed in literature could be implemented and evaluated. Better synchronization schemes, like Lookup Transient message and Rollback may be evaluated. The work distribution has been approximately equal with Kaushik handling the Gossip protocol implementation and evaluation and Pushkar and Parminder handling the framework setup and creation, partitioning and simulation of the Parallel Scale Free Network simulation.

References

[1] CStar. http://c-starsolutions.com/starcustomers, 2013.

[2] Robert S. Pienta and Richard M. Fujimoto. On the parallel simulation of scale-free networks. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '13, pages 179–188, New York, NY, USA, 2013. ACM.

[3] Benoit Garbinato, Denis Rochat, and Marco Tomassini. Impact of scale-free topologies on gossiping in ad hoc networks. *IEEE NCA*, 58(301):1–10, March 2007.

[4] Angelo Gabriele D and Stefano Ferretti. Simulation of scale-free networks. *ACM - Journal*, 58(301):1–10, March 2009.

[5] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

[6] George Karypis and Vipin Kumar. Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, SIAM Journal on Scientific Computing, 1995.

[7] Lisandro Dalcín, Rodrigo Paz, Mario Storti, and Jorge D'Elía. Mpi for python: Performance improvements and mpi-2 extensions. *J. Parallel Distrib. Comput.*, 68(5):655–662, May 2008.

[8] Sunil Thulasidasan, Lukas Kroc, and Stephan Eidenbenz. Developing parallel, discrete event simulations in python: First results and user experiences with the simx library. 4^{th} *International Conference Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2014)*, June 2014.

[9] Xufei Wang, Huan Liu, Peng Zhang, and Baoxin Li. Identifying information spreaders in twitter follower networks. Technical Report TR-12-001, School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ 85287, USA, 2012.

[10] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection, June 2014.