Packet level simulator of **elastic caching and service chain** network : outline

This is a **pull-based** system

**Objects:**

Agents:

-node

-caches (Note: ``item’’ in cache is a tuple ``(item, stage)’’ in the network)

-computing units (CU)

-links

-demand (continuously making requests)

Methods:

-flow estimation

-cost calculation (flow)

-link marginal estimation (flow)

-cache marginal estimation (flow)

-self marginal calculation (flow)

-packet delay measurement

-message pushing (3 different messages)

-cache/routing reshuffle (+ rounding)

-request routing table

-cache size determining

Messages:

-Request msg (for different stages, for stage 0, request for data)

-Respond msg (for stage 0, carries data)

-Broadcast msg (pT/pt)

Pipelines:

-node: requester (push requests)

-node: request input queue + request router

-node: respond input queue + item router

-node: computing unit input queue

-link: for requests (with delay)

-link: for responds (with delay)

-link: represent computation (with delay)

-link: for broadcasts (with delay)

(note: these links are separate, since request msg has 0 size, and broadcast msg are sent in separate channels)

Global objects:

-graph

Auto Process:

-Time

-Requesting

-Updating

-Monitoring

Compare:

-cache replacement:

-cache size:

-computation placement:

-joint: deco+service chain, Jianan+caching

Changes from Startis Code：

-LRU etc. Should update according to respond msg, not request msg.

**Behavior description:**

- This system is a pull-based system, working on a finite set of ``items’’ and ``applications’’. Each application is a service chain. We say a ``stage’’ is a tuple (a,k), where a is the application and k is the position of chain. K starts from 0.

- Each demand is assigned a fixed tuple of (id, instance #, node, item, stage, rate), representing that ``node’’ continuously making requests for the computation result for ``item’’ of ``stage’’ with that rate, Poisson process. (the instance # gives every request instance a unique number to distinguish, could re-cycle after a while)

- Each demand is realized by a requester pipeline, treated as an in-going link.

- The requests coming out all in-going links are all merged to the request input pipeline.

- Request router pulls requests from request input pipeline.   
 First see if the (item,stage) tuple is cached, if so, discard the request msg, generate a respond msg and send back to the in-going link (reversed direction).

If not cached, router decide whether to compute locally, or to forward.

If to compute (probability phi\_i0), put the request in the computing input queue, and go through the computation link to transfer to stage k+1.

If to forward, **randomly pick** an out-link by the distribution (\phi\_ij(k)) and forward

(here could have other method that achieves the specified fraction).

After forwarding, keep a record of tuple (id, instance #, receiving from, forward to), used to route the respond later.

- The responses coming out all in-going links are all merged to the response input pipeline.

- Response router pulls responses from the response input pipeline, check for instance#, and send backwards according to the record (id, instance #, receiving from, forward to).

- To make the actual fraction as close to theoretical possible, the caching strategy is kept in each time slot, which is chosen from probability (y\_i(k)). The caching strategy is re-picked according to the same set of y\_i(k), for a period (a number of slots, say 10 slots). Probability y\_i(k) and \phi\_ij(k) are updated every 10 slots, which is called a ``update slot’’. The update slot is at the end of a period.

- Note that when picking random cached items, keep the total cache size unchanged. Use the method by adaptive caching paper.

- To deliver the messages, request msg are sent through request pipeline. Response msg are sent through response pipeline. Broadcasts are sent through broadcast pipeline.

Measurements are collected separately from different type of pipelines.

- Pipelines for links will cause delay. This delay is not uses by nodes’ strategies, but for system measurements only.

- Between two update slots, the network collect and average statistics.

Each link measures the flow rate (note: each msg could have different size), the link marginal, and the link cost.

Each cache measures the cache size, cache cost and cache marginal.

Note: the measure score are not collected in the update slot, to be fair.

-At the beginning of the update slot, broadcast msgs are sent. This is initialized from the end of paths for each k, i.e., nodes with \phi\_ij(k) =0 but y\_im(k) \neq 0.

Sending of these ctrl msg have an associated id number, to distinguish from previous update slots. The next id update msg should come after receiving all downstream of current id.

For each item , node i keeps a record for the list of receiving broadcast msgs from. When the list meets all downstream nodes (j with phi\_ij(k) > 0), node i calculates its own pT/pr, and sent to all its neighbor nodes. (Not only send to upstream nodes, since when running GP, node i needs to know D’ji(Fji)+pTpr\_j for all j. The ctrl msg payloads carrying D’ji(Fji)+pTpr\_j should also be recorded for all received j. )

Note: we assume the broadcast could finish within a update slot.

- Blocked Nodes:

The network operator chooses whether the loops are forbidden or not. If choose to allow loops, no blocked nodes will be practiced. If not, the following blocked nodes mechanism:

At the beginning of each update slot (right after sending the ctrl msgs), the network runs a blocked nodes update function (centralized), to calculate the blocked nodes and assign to each node.

The blocked node update function does the following:

1. For each item k, gathers the global \phi\_ij(k) and generates a DAG, called G\_k. (Note that if G\_k is not a DAG, this means the previous solution is not loop-free, will send errors)
2. For each k, topo-sort G\_k and attain a total order of all nodes.
3. For each k and each node i, set the blocked node set to j >= i in the total order of G\_k.

- At the end of the update slot, each node is informed with \delta\_i(k). Use a **gradient projection** (could be further update to scaled gradient projection) to individually calculate the new strategy.

- Note: to ensure that the ratio of routing stepsize and caching stepsize can effectively control the ratio of variable changing speed (e.g., if caching stepsize << routing stepsize, the algorithm will converge fast to the optimal routing, while the caching variable changes slowly), - The amount of non-opt out-variable is collected for all non-opt routing all caching.

- After collecting non-opt sum, equally distribute fraction (alpha/ (alpha + beta)) of the sum to opt routing vars and (beta/(alpha + beta)) of the sum to opt caching vars.

- Using this scheme, if beta << alpha, the caching variables will not be decreasing much (since the non-opt sum collecting with stepsize) and not increasing much (since only beta/(alpha + beta) of the sum is assigned to caching)

- If the routing var is already optimal but the caching delta is lower, the non-optimal sum is collected from all neighbors, and then distributed back to all neighbors.

Each node keeps the strategy for the next period (10 slots).

- Variable update: we assume the convergence speed of caching variable should be much slower than the routing variable. To achieve so, could either 1) set the stepsize of y is much smaller than of phi, or 2) update the caching variable one node at a time.

- Baseline routing

Shortest path.

- Baseline caching - simple caches (LRU, LFU, FIFO, RR):

LRU: The cache maintains a sorted array of the items, sorted by the visit time. Cache items according to the **respond msgs** that latest passes through. If cache is full, discard the earliest cached item.

LFU: The cache maintains a sorted list of tuple (item, # of responses passing through), sorted by the usage number. Cache items at the top of the list.

FIFO: The cache maintains a FIFO queue of responses passing through. If cache hit, do nothing.

- Baseline caching - optimized caches (LMIN, ??):

LMIN: Adaptive caching paper with routing.

??

- Baseline cache size - heuristic

Miss Rate: incremental add cache from 0, to the node with highest cache miss count.

LMIN+Global cache budget: increasing the total cache budget, until the total cost is no longer decreasing.

**Simulation Parameters:**

(Format: - Name, Type, Default value, Description)

- ConfigFileName, String, ‘config.txt’,

The configuration file name that stores all other parameters, so that when run, only this one parameter is needed to be input.

- IsReadScenario, bool, False,

If False, read all other parameters and generate new scenario, save as ‘SavedScenario’.

If True, discard all generating parameter, read scenario from parameter ‘SavedScenario’.

- SavedScenario, string, ‘TestScenario.txt’

Saved scenario settings. Including: topology (nodes and links), cost functions, request patterns, ...

Note that the saved scenario only guarantees statistics, not the realization of random events, like the exact time of request msgs are generated, despite the rate is the same.

- GraphType, string, ‘Connected\_ER’, topology type

- N, int, 20, network size (number of nodes)

- K, int, 20, catalog size (number of items)

- M, int, 2, number of cache types

**Module Detail:**

- G: the network topology, networkx.DiGraph

- G[i][j][‘LinkPara’]: link cost parameter, uniformly from [LinkParaMin,LinkParaMax]

- G.nodes[i][‘CachePara’]: cache cost parameter, uniform

- DesServerList: List of designated servers, assume each item has one server

- Demands: list of tuple (id, node, item, rate)

- CacheNet.DesServers: dictionary of designated servers. {item: [list of servers]}

- CacheNet.RequestRates: dictionary of request rates. {node:item: rate}

- self.Nodes[i]['RequestPipe']: request input queue of node i, containing r\_i(k) and \phi\_ji(k)

- self.RouteVarInit[i][k]: the array of phi\_i\*(k). The j-th element is phi\_ij(k)

- Each node is equipped with 1 real cache. The cache size of cache types is virtually assigned.

**Benchmarks:**

The algorithm (general case) is compared with a number of other algorithms.

For now, we always assume M=1, and no need for cache type optimization (the parameter ‘CacheTypeAlgo’ is always set to ‘All-first’)

1. Compare with the proposed (fixed routing special case) algorithm:

self.AlgoConfig.CacheAlgo == 'GCFW' \

and self.AlgoConfig.RouteAlgo == 'ShortestPath-iter' \

and self.AlgoConfig.SizeAlgo == 'GCFW'\

and self.AlgoConfig.CacheTypeAlgo == 'All-first':

A centralized algorithm presented for the fixed routing special case, combined with shortest path routing. When using this benchmark for fixed-routing special case, the topology should be set to ‘Tree’.

GCFW is also implemented in time slots. Totally N slots, with epsilon = N^-3.

GCFW starts with zero-cache and shortest path. At the end of a time slot, a centralized process calculates the gradients of function A(Y) and B(Y), the detailed calculation expression is presented in the paper.

To carry out this calculation, at the initialization, the process would calculate and record the ‘paths’, i.e., p\_vk, for all v and k, to speed up the following iterations.

During the process, the objective value (theoretical) and the variables are all saved. After N iteration, the process choose the best among historical solutions.