Randomized Sets and Multisets

A Literate C++ Program

Andreas Klappenecker and Hyunyoung Lee

October 26, 2004

1 Quorum

Suppose that we have n servers which we number from 0 to $n-1$. A client process selects a quorum of k servers uniformly at random. If the quorum size k is larger than $n/2$, then any two quorums have a server in common; this property allows us to design a protocol that ensures consistency even if we have multiple writing processes that access data on the n servers. In a more recent development, Malkhi et al. observed that one can choose a significantly smaller quorum size k, say $k = \Omega(n^{1/2+\varepsilon})$, and still get a nontrivial intersection with high probability when n is large.

Class Declaration. The class `quorum` implements a quorum of size k among n servers with a k-subset Q of $\{0, \ldots, n-1\}$. The class provides a method to construct such a quorum `W` by `quorum W(n,k)`. The file `quorum.h` contains the complete interface:

1 ⟨*quorum.h* 1⟩≡

```
#ifndef QUORUM_H
#define QUORUM_H
#include <set>
typedef unsigned u_int;

class quorum {
  u_int n;
  u_int k;
  std::set<u_int> Q;
public:
  quorum() { n=0; k=0; };
  quorum(u_int, u_int);
  ~quorum() { Q.clear(); }
```

```
  quorum(const quorum& rhs);
  quorum& operator=(const quorum& rhs);
  u_int num()  const { return n; }
  u_int size() const { return k; }
  void num(u_int nn) { n = nn; }
  void size(u_int kk) { k = kk; }
  std::set<u_int> read() const { return Q; }
  std::set<u_int> choose();
};

std::ostream& operator<<(std::ostream& os, quorum& qrm);
#endif
```

The accessor functions `num` and `size` allow us to read the number of servers `n` and the quorum size `k`, respectively. You can use `W.num(5)` to change the number `n` of servers in `W` to 5. Similarly, `W.size(3)` sets the quorum size `k` to 3.
We can obtain the set constituting the current quorum by `W.read()`. An important method is `choose()` that allows us to choose a new quorum with `k` elements uniformly at random from the set $\{0, \ldots, n-1\}$. The output stream operation `<<` is overloaded to provide a convenient method to print the current quorum.

Member Functions. The file `quorum.cc` contains the member functions of the class `quorum`. The constructor `quorum(n,k)` with two parameters assigns the number of servers `n` and quorum size `k`, and then chooses uniformly at random a quorum of `k` servers. The random number generator is seeded with the current time so that the behavior of the quorum selection appears to be random.

2 ⟨*quorum.cc* 2⟩≡ 3a ▷

```
#include <cstdlib>
#include <ctime>
#include <iostream>
#include "quorum.h"
using namespace std;

quorum::quorum(u_int nn, u_int kk) {
  n = nn; k = kk;
  static int seen = 0;
  if(!seen) { srand(static_cast<unsigned>(time(0))); seen = 1; }
  choose();
}
```

The next member function is a copy constructor that allows us to define a quorum `A` as a copy of another quorum, `quorum A = Q(10,3)`.

3a ⟨*quorum.cc* 2⟩+≡ ◁2 3b▷

```
quorum::quorum(const quorum& rhs) {
  static int seen = 0;
  if(!seen) { srand(static_cast<unsigned>(time(0))); seen = 1; }
  n = rhs.num();
  k = rhs.size();
  Q = rhs.read();
}
```

We can also assign the content of one quorum to another quorum; we take care that a nonsensical `Q=Q` assignment does not free the data in the quorum.

3b ⟨*quorum.cc* 2⟩+≡ ◁3a 3c▷

```
quorum& quorum::operator=(const quorum& rhs) {
  if(this != &rhs) {
    n = rhs.num();
    k = rhs.size();
    Q.clear();
    Q = rhs.read();
  }
  return *this;
}
```

The method `choose()` provides the function which selects a quorum of size `k` uniformly at random. Suppose that `W` contains a quorum $Q = \{1, 2, 5\}$ of $k = 3$ servers from $n = 10$ servers. Then `W.choose()` could select, for example, the new quorum $Q = \{2, 7, 9\}$.

3c ⟨*quorum.cc* 2⟩+≡ ◁3b 4a▷

```
set<u_int> quorum::choose() {
  for( Q.clear() ; Q.size() != k; Q.insert(rand() % n));
  return Q;
}
```

The file also contains the definition of the overloaded operator <<. For example, we can print the current state of the quorum W with cout << Q;. In our example above, the result would be {2,7,9}.

4a ⟨*quorum.cc* 2⟩+≡ ◁3c

```
ostream& operator<<(ostream& os, quorum& qrm) {
  const set<u_int> W = qrm.read();
  os << "{";
  for(set<u_int>::const_iterator i=W.begin(); i!=W.end(); i++) {
    if( i != W.begin() ) { os << ","; }
    os << *i;
  }
  os << "}";
  return os;
}
```

The procedure simply iterates over the set Q and prints the value of the elements separated by commas.

Example A. Let's illustrate the usage of the quorum class with a simple example.

4b ⟨*testA.cc* 4b⟩≡

```
#include <iostream>
#include "quorum.h"
using namespace std;

int main() {
  quorum Q(10,5);    // choose a quorum of 5 out of 10 servers
  cout << Q << endl; // print this quorum
  Q.choose();        // choose a new quorum
  cout << Q << endl; // print again
}
```

The example chooses uniformly at random 5 out of 10 servers, prints this quorum, chooses a new quorum, and prints again. The result is, say,

```
{1,6,7,8,9}
{2,3,5,8,9}
```

Please note that different runs may yield different results.

Example B. In some cases, you might want to increase the quorum size to increase the accuracy of subsequent operations.

4c ⟨*testB.cc* 4c⟩≡

```
#include <iostream>
#include "quorum.h"
using namespace std;

int main() {
  quorum Q(10,5);    // choose a quorum of 5 out of 10 servers
  cout << Q << endl; // print this quorum
  Q.size(6);         // increase quorum size to 6 servers
  Q.choose();        // choose a new quorum
  cout << Q << endl; // print again
}
```

A sample output is given by

```
{1,3,4,6,9}
{1,5,6,7,8,9}
```

Summary. It is helpful to keep the following properties of the quorum class in mind:

- The read operation does not change the quorum, so `Q.read()` followed by `Q.read()` yields the same result unless the quorum is explicitly changed between the two read operations.
- You can choose a quorum uniformly at random by `Q.choose()`.
- If A and B are two quorums of k out of n servers, then the probability that the two quorums have trivial intersection is given by

$$\Pr[A \cap B = \emptyset] = \binom{n-k}{k} \Big/ \binom{n}{k} \leq e^{-k^2/n}.$$

The main use of quorums will be in randomized set operations, which we explain in the next section.

2 Randomized Sets

A randomized set is a distributed shared data structure which approximates the behavior of a set. Consider a set M that is represented on n servers by replicas M_r with $0 \leq r < n$. The replicas M_r are subsets of M such that their union $\bigcup M_r = M$. The main idea is that a client process accesses a quorum of k replicas for any set operation.

The class `rset` allows you to experiment with this randomized data structure. The class provides member functions for all operations that are described in the companion paper. An object S of type `rset<T>` is determined by (i) a vector of size n that contains the replicas of the set S, and (ii) a quorum object `Q` that contains the number of servers n, the quorum size k, and the quorum chosen by the last operation.

Class Declaration. Let's start with an overview of the `rset` class. We declare and define all member functions of the class `rset` in the file `rset.h` to avoid issues with the separate compilation of template classes. The file `rset.h` is organized as follows:

6 ⟨*rset.h* 6⟩≡

```
#ifndef RSET_H
#define RSET_H
#include <cmath>
#include <ctime>
#include <iostream>
#include <vector>
#include <set>
#include <iterator>
#include <algorithm>
#include <functional>
#include "quorum.h"
typedef unsigned int u_int;
typedef std::set<u_int> ui_set;
⟨rset class declaration 7⟩
⟨rset definitions 8⟩
#endif
```

The organization of this file is straightforward. We simply include all standard library header files that we need and define an unsigned integer type `u_int` and a type `ui_set` for sets of unsigned integers. The main purpose is the definition of a class `rset<T>` that represents a randomized set of type `T`. Let's have a look at the class declaration first:

7 ⟨*rset class declaration* 7⟩≡ (6)

```
template <class T>
class rset {
  std::vector< std::set<T> > replica;
  quorum Q;
public:
  // Constructors, destructor, assignment
  rset() { }
  rset(u_int, u_int);
  rset(const rset<T>&);
  ~rset();
  const rset<T>& operator=(const rset<T>&);

  // Accessors and mutators
  u_int num() const    { return Q.num();  }
  u_int size() const   { return Q.size(); }
  void size(u_int k)   { Q.size(k);       }
  quorum qrm() const;
  std::set<T> read(u_int) const;

  // Basic rset operations
  std::set<T> read();
  void insert(const T);
  void insert(const std::vector<T>&);
  rset<T> operator+(rset<T>&);
  rset<T> operator*(rset<T>&);
  rset<T> operator-(rset<T>&);
};

// Related procedures
template <class T>
std::ostream& operator<<(std::ostream&, std::set<T>&);
template <class T>
std::ostream& operator<<(std::ostream&, rset<T>&);
```

An object of type `rset<T>` contains a vector `replica` of n sets of type `set<T>` and a quorum `Q` of k out of n servers. The set `replica[i]` represents, as the name suggests, the replica on server `i`.

The class provides member functions to create and destruct randomized sets. For example, if you want to create a randomized set `S` of double precision floating point numbers on 10 servers and the default quorum size is 4, then you can create this randomized set by `rset<double> S(10,4)`. You can create another randomized set with the same parameters by `rset<double> T = S`.

The replicas are initially empty. You can add an element, say `3.14`, to the randomized set `S` by `S.insert(3.14)`; the member function will choose 4 servers uniformly at random and insert the element `3.14` into the replicas of these servers. The operation `S.read()` simulates the read operation of a client and returns the union of 4 replicas `S.replica[i]` that are chosen uniformly at random. The union, intersection and set difference operation on randomized sets are respectively given by `S+T`, `S*T` and `S-T`.

Member Functions. We describe now the member functions of the class `rset` in more detail. We recommend that you briefly skim through this paragraph to get familiar with the basic member functions and then experiment with the examples given in the subsequent paragraphs. You can then refer back to this paragraph to get more detailed explanations.

> For easier navigation, we use the symbol ♠ for constructors and destructors, ♢ for accessor and mutator functions, ♣ for randomized set operations, and ♡ for related procedures.

♠ *Constructor.* The constructor of an `rset` initializes the number of servers `n` and the quorum size `k` used by the operations on `rset`. The `n` replicas are initialized to empty sets. If you want to define a randomized set `S` of integers with 50 replicas and quorum size 16, then you can do that by `rset<int> S(50,16)`.

8 ⟨*rset definitions* 8⟩≡ (6) 9a▷

```
template <class T>
rset<T>::rset(u_int n, u_int k) {
  static int seen = 0;
  if(!seen) { srand(static_cast<unsigned>(time(0))); seen = 1; }
  for(u_int i=0; i<n; i++) {
    replica.push_back(std::set<T>() );  // replica[i] = {}
  }
  Q.num(n);
```

```
    Q.size(k);
    Q.choose();
  }
```

♠ *Copy Constructor.* A second method to construct a randomized set `S2` starts with a given randomized set `rset<T> S1(n,k)` and is realized by `rset<T> S2 = S1`. As you might have guessed, the quorum and the replicas are copied.

9a ⟨*rset definitions* 8⟩+≡ (6) ◁8 9b▷

```
  template <class T>
  rset<T>::rset(const rset<T>& rhs) {
    Q = rhs.qrm();
    for(u_int i=0; i<Q.num(); i++) {
      replica.push_back(rhs.read(i));
    }
  }
```

♠ *Destructor.* The destructor `~rset` of a randomized set simply clears the vector of replicas and the quorum.

9b ⟨*rset definitions* 8⟩+≡ (6) ◁9a 9c▷

```
  template <class T>
  rset<T>::~rset() {
    replica.clear();
    Q.read().clear();
  }
```

♠ *Assignment.* Suppose that you have defined random sets `A` and `B` of type `T`. The next member function allows you to perform the assignment `A=B`. A typical usage is in expressions such as `A=B*C`.

9c ⟨*rset definitions* 8⟩+≡ (6) ◁9b 10a▷

```
  template <class T>
  const rset<T>& rset<T>::operator=(const rset<T>& rhs) {
    if (this != &rhs) {
      Q = rhs.qrm();
      replica.clear();
      for(u_int i=0; i<Q.num(); ++i) {
        replica.push_back(rhs.read(i));
      }
    }
    return *this;
  }
```

♢ *Read Replica.* Suppose that you have a randomized set `S`. If you want to know what elements are contained in the replica of `S` on server `x`, then you can access this set by `S.read(x)`.

10a ⟨*rset definitions* 8⟩+≡ (6) ◁9c 10b▷

```
template <class T>
std::set<T> rset<T>::read(u_int x) const {
  return replica[x];
}
```

♢ *Quorum.* If you are interested in the quorum that has been used by the last insert or read operation of a randomized set `S`, then you can access this quorum by `S.qrm()`.

10b ⟨*rset definitions* 8⟩+≡ (6) ◁10a 10c▷

```
template <class T>
quorum rset<T>::qrm() const {
  return Q;
}
```

♣ *Insert.* The next member function chooses a write quorum `W`, and inserts the element to the `k` replicas indexed by `W`. For example, if we have a randomized set of doubles, `rset<double> S(50,16)`, then `S.insert(3.14)` adds the value 3.14 to 16 replicas of `S`. You can access the write quorum that has been chosen by this operation by `S.qrm()`.

10c ⟨*rset definitions* 8⟩+≡ (6) ◁10b 11a▷

```
template <class T>
void rset<T>::insert(const T elem) {
  Q.choose();
  ui_set W = Q.read();
  for(ui_set::const_iterator it = W.begin(); it != W.end(); ++it) {
    replica[*it].insert(elem);
  }
}
```

♣ *Insert multiple elements.* Suppose you want to simulate a client that adds several elements to a randomized set `S` of type `T`. If the elements are represented by a vector `v` of type `vector<T>`, then you can use `S.insert(v)` as an abbreviated form for `S.insert(v[0])`, `S.insert(v[1])`, ...

11a ⟨*rset definitions* 8⟩+≡ (6) ◁10c 11b▷

```
template <class T>
void rset<T>::insert(const std::vector<T>& vec) {
   typename std::vector<T>::const_iterator it;
   for(it = vec.begin(); it != vec.end(); it++) {
     insert(*it); // insert element by element
   }

}
```

♣ *Read.* A read operation chooses a quorum `R` of `k` servers. It returns the union of the `k` replica sets indexed by the read quorum `R`.

11b ⟨*rset definitions* 8⟩+≡ (6) ◁11a 11c▷

```
template <class T>
std::set<T> rset<T>::read() {
  Q.choose();
  ui_set R = Q.read(); // get a read quorum
  std::set<T> S;
  ui_set::const_iterator it;
  for(it = R.begin(); it != R.end(); ++it) {
    S.insert(replica[*it].begin(), replica[*it].end());
  }
  return S;
}
```

♣ *Union.* Suppose you have three randomized sets `A`, `B`, and `C` of type `T`. The operation `C=A+B` provides a randomized union operation. This operation simulates a client that performs `A.read()` and `B.read()`, takes the union of these two sets, clears the replicas of `C`, and then inserts the elements `A.read()`∪`B.read()` into `C`.

11c ⟨*rset definitions* 8⟩+≡ (6) ◁11b 12a▷

```
template <class T>
rset<T> rset<T>::operator+(rset<T>& B) {
  std::set<T> Ar = read(), Br = B.read();
  std::vector<T> C;
  set_union(Ar.begin(),Ar.end(),
            Br.begin(),Br.end(),back_inserter(C));
  rset<T> R(num(),size());
  R.insert(C);
  return R;
}
```

♣ *Intersection.* Suppose you have three randomized sets `A`, `B`, and `C` of type `T`. The operation `C=A*B` provides a randomized intersection operation. This operation simulates a client that performs `A.read()` and `B.read()`, takes the intersection of these two sets, clears the replicas of `C`, and then inserts the elements `A.read()`∩`B.read()` into `C`.

12a ⟨*rset definitions* 8⟩+≡ (6) ◁11c 12b▷

```
template <class T>
rset<T> rset<T>::operator*(rset<T>& B) {
  std::set<T> Ar = read(), Br = B.read();
  std::vector<T> C;
  set_intersection(Ar.begin(),Ar.end(),
                   Br.begin(),Br.end(),back_inserter(C));
  rset<T> R(num(),size());
  R.insert(C);

  return R;
}
```

♣ *Set Difference.* Suppose you have three randomized sets `A`, `B`, and `C` of type `T`. The operation `C=A-B` realizes `A.read()`\ `B.read()` and assigns the result to `C`. Warning: The probabilistic nature of the read operations can lead to elements in this set that do not belong to `A`\`B`. The operation should be used with care.

12b ⟨*rset definitions* 8⟩+≡ (6) ◁12a 13a▷

```
template <class T>
rset<T> rset<T>::operator-(rset<T>& B) {
  std::set<T> Ar = read(), Br = B.read();
  std::vector<T> C;
  set_difference(Ar.begin(),Ar.end(),
                 Br.begin(),Br.end(),back_inserter(C));
  rset<T> R(num(),size());
  R.insert(C);
  return R;
}
```

♡ *Print sets.* We overload the operator `<<` so that we can print a set with elements of type `T` in the usual form, e.g., `{1,2,3}`.

13a ⟨*rset definitions* 8⟩+≡ (6) ◁12b 13b▷

```
template <class T>
std::ostream& operator<<(std::ostream& os, std::set<T>& S) {
  typename std::set<T>::const_iterator it;
  os << "{";
  for(it=S.begin(); it != S.end(); it++) {
    if(it != S.begin() ) { os <<","; }
    os << *it;
  }
  os << "}";
  return os;
}
```

♡ *Print randomized sets.* Finally, we provide a procedure to print the replicas of a randomized set.

13b ⟨*rset definitions* 8⟩+≡ (6) ◁13a

```
template <class T>
std::ostream& operator<<(std::ostream& os, rset<T>& S) {
  for(u_int r=0; r<S.num(); r++) { // iterate through replicas
    os << "Server " << r << ": ";
    std::set<T> Rep = S.read(r);
    os << Rep << endl;
  }
  return os;
}
```

Example C. Let's illustrate the randomized set class with some simple examples. In the first example, we construct a randomized set of integers that is replicated over 4 servers. We insert the elements $1, 2, 3, 4$, and 5 using quorum size 2, and print the state of the replicas so that one can appreciate the construction of this randomized set.

14 ⟨*testC.cc* 14⟩≡

```
#include <iostream>
#include "quorum.h"
#include "rset.h"
using namespace std;

int main() {
  rset<int> S(4,2);
  for(int i=1; i<6; i++) {
    cout << "Adding element " << i << " yields:" << endl;
    S.insert(i);
    cout << S;
  }
}
```

Example D. The size of a quorum influences the results significantly. For example, the following program illustrates the effect of the quorum size in terms of the read operations. We first construct a set of cardinality 300 using quorums of size 16 and 50 replica servers. We read 15 times and note the cardinalities of the sets. We then repeat the same experiment with a reduced quorum size of 12 servers (which is too small).

15 ⟨*testD.cc* 15⟩≡

```
#include <cstdlib>
#include <ctime>
#include <iostream>
#include "quorum.h"
#include "rset.h"
using namespace std;

void print_sizes(int k) {
  rset<int> S(50,k);
  for(int i=0; i<300; S.insert(i++));

  for(int i=0; i<14; i++) {
    cout << S.read().size() << ", ";
  }
  cout << S.read().size() << "." << endl;
}

int main() {
  cout << "Construct a set of cardinality 300.";
  cout << " For quorum size k=16,";
  cout << endl << "repeated read operations yield ";
  cout << "sets of cardinality:" << endl;
  print_sizes(16);

  cout << "For quorum size k=12, we obtain sets ";
  cout << "of cardinality: " << endl;
  print_sizes(12);
}
```

Example E. We have restricted ourselves so far to the creation of a randomized set and subsequent read operations. Using these primitives, we can define probabilistic versions of set operations such as union, intersection, and set difference in the following way:

$$\texttt{A.read()} \cup \texttt{B.read()}, \quad \texttt{A.read()} \cap \texttt{B.read()}, \quad \texttt{A.read()} \setminus \texttt{B.read()}.$$

The read operation `A.read()` is expected to return $(1-\varepsilon)|\texttt{A}|$ elements, where ε is determined by the number of servers n and the quorum size k,

$$\varepsilon = \binom{n-k}{k} \Big/ \binom{n}{k}.$$

This means that $A \cup B$ and $A \cap B$ are approximated by subsets. However, $A \setminus B$ is approximated by a set that can be too large or too small (and we discourage the usage of this operation). Anyway, you might want to gain experience with these operations so that you know what to expect.

16 ⟨*testE.cc* 16⟩≡

```
#include <iostream>
#include <set>
#include <vector>
#include "rset.h"
using namespace std;

int main () {
  int n = 6;
  int k = 3;
  rset<int> A(n,k);
  rset<int> B(n,k);
  A.insert(1);
  A.insert(2);
  A.insert(3);

  B.insert(3);
  B.insert(4);
  B.insert(5);

  cout << "A = {1,2,3} " << endl;
  cout << "B = {3,4,5} " << endl;

  cout << "A union B = {1,2,3,4,5}" << endl;
  rset<int> C = A + B;
  cout << C;
```

```
  cout << "A intersection B = {3}" << endl;
  rset<int> D = A * B;
  cout << D;

  cout << "A minus B = {1,2}" << endl;
  rset<int> E = A - B;
  cout << E;
}
```

Example F. In the previous two examples, we have studied the effect of the quorum size on the read operations. We conclude this section by giving a small program that allows you to calculate the probability ε that two quorums will be disjoint, $\varepsilon = \binom{n-k}{k}/\binom{n}{k}$.

We prompt the user to provide the quorum size k and the number of servers n. The calculation is done with a procedure `binom` that calculates the binomial coefficient $\binom{n}{k}$. The type `T` of the template should be some integer type such as `int`, `long` or the like.

17 ⟨*binomial* 17⟩≡ (18)

```
template <class T>
T binom(T n, T k) {
  T mi = min<T>(k,n-k);
  if (mi < 0)
    return 0;
  else if (mi == 0)
    return 1;
  else {
    T ma = max<T>(k,n-k);
    T res = ma+1;
    for (T i = 2; i <= mi; i++) {
      res = (res*(ma + i)) / i;
    }
    return res;
  }
}
```

The calculation of the binomial coefficient rests on the observation that the product $(n-k+1)(n-k+2)\cdots(n-k+\ell)$ is divisible by $\ell!$ for $\ell \geq 1$. Therefore, it is possible to calculate $\binom{n}{k} = (n-k+1)(n-k+2)\cdots n/k!$ by successive multiplications of the terms $(n-k+i)$ and divisions by i that keep the intermediate products relatively small.

18 ⟨*testF.cc* 18⟩≡

```
#include <iostream>
#include "rset.h"
using namespace std;

⟨binomial 17⟩

int main() {
  int n,k;

  // prompt user to provide the key parameters
  cout << "Please provide the number of replicas ";
  cout << "n = ";
  cin >> n;
  cout << "the quorum size ";
  cout << "k = ";
  cin >> k;

  double eps = binom<long long>(n-k,k)/(double)binom<long long>(n,k);
  cout << "Precision epsilon = " << eps << endl;
}
```

3 Conclusions

Randomized sets are a natural data structure for distributed algorithms that share information among different servers. If the number n of servers is small, then it is natural to use a quorum size of $k > n/2$ to access the replicas. However, for a modest to larger number of servers, the message complexity for such a traditional quorum approach can be intolerable. The probabilistic approach outlined in the paper "Randomized Sets and Multisets" by H. Lee and A. Klappenecker, 2004, can provide a remedy. We encourage you to experiment with our programs, so that you are able to appreciate the advantages and disadvantages of this approach.

It is somewhat surprising that a systematic study of randomized data structures is still lacking. Our approach generalizes directly to the other associative containers of Stepanov's STL (namely multisets, maps, and multimaps). We encourage the reader to modify our programs to study such randomized data structures (this is nice exercise that is not difficult). We would be delighted to hear about implementations of other randomized data structures.

Acknowledgments. We thank Gabriel Dos Reis and Bjarne Stroustrup for very helpful advice on compiler and language issues. The research by H.L. was supported by University of Denver PROF grant 88197. The research by A.K. was supported by NSF CAREER award CCF 0347310, NSF grant CCR 0218582, a TEES Select Young Faculty award, and a Texas A&M TITF initiative.